Introdução à Ciência da Computação Usando a Linguagem C

Editado por:
Carlos Hitoshi Morimoto
Ronaldo Fumio Hashimoto

Compilação de notas de aulas
utilizadas em disciplinas introdutórias
do Departamento de Ciência da Computação
do Instituto de Matemática e Estatística
da Universidade de São Paulo.

São Paulo – 2010

Prefácio

Esta apostila é uma compilação de notas de aulas de disciplinas de Introdução à Ciência da Computação oferecidas pelo Departamento de Ciência da Computação do IME/USP e tem por objetivo apresentar conceitos básicos de computação e programação por meio de exercícios práticos.

Originalmente, a compilação destas notas de aulas foi realizada com o objetivo de apresentar um material de apoio para o curso não-presencial de MAC2166 oferecido aos alunos da Escola Politécnica da USP. Tradicionalmente, a linguagem C é utilizada no ensino dessa disciplina, e por isso ela também é utilizada nessa apostila. Com este material pronto, pensamos que ele possa ser útil também aos alunos de cursos presenciais.

Pela nossa experiência, o método mais eficaz para aprender a programar é programando. Por isso, as notas de aula estão organizadas na forma de resolução de vários exercícios, com foco na resolução de problemas, estruturando a solução na forma de um programa, ao invés da simples descrição de comandos da linguagem C.

Obviamente, cada professor confere à sua aula um "sabor" e um "colorido" diferente: cada um tem sua própria maneira de ensinar. Por isso, advertimos que não necessariamente estas notas de aulas serão seguidas pelo seu professor, mas de toda maneira, esperamos que esta apostila seja útil para os alunos como material de estudo e de apoio em seu primeiro contato com um curso introdutório de computação e programação.

Vale a pena ressaltar que estas notas de aula são resultado de uma experiência e de um esforço conjunto de muitos professores do departamento que já ministraram estas disciplinas. Por isso, queremos deixar aqui nosso reconhecimento deste esforço e também nossos agradecimentos.

Um bom estudo!

São Paulo, janeiro de 2010.

Carlos Hitoshi Morimoto e Ronaldo Fumio Hashimoto

Sumário

1 Como Funciona um Computador

Carlos H. Morimoto e Thiago T. Santos

O objetivo dessa aula é apresentar uma breve descrição do funcionamento de um computador para que você possa se familiarizar com alguns termos que serão muito utilizados ao longo do curso. Para descrições mais completas, você pode encontrar na Internet vários sítios que descrevem o funcionamento dessa máquina, como por exemplo `http://www.ime.usp.br/~macmulti/historico/`.

Ao final dessa aula você deve ser capaz de:

- Descrever as principais partes de um computador: Unidade Central de Processamento (microprocessadores), Memória e Dispositivos de Entrada e Saída.
- Descrever o que é software e hardware.
- Descrever o que são linguagens de alto nível e de baixo nível.
- Definir o que é um compilador.
- Descrever o que é um algoritmo e dar exemplos práticos.

1.1 As partes de um computador

A maioria dos computadores segue a arquitetura de von Neumann, que descreve um computador como um conjunto de três partes principais: a unidade central de processamento ou UCP (que por sua vez é composta pela unidade lógico-aritmética (ULA) e pela unidade de controle (UC)), a memória e os dispositivos de entrada e saída (E/S). Todas as partes são conectadas por um conjunto de cabos, o barramento. Esses componentes podem ser vistos na figura 1a.

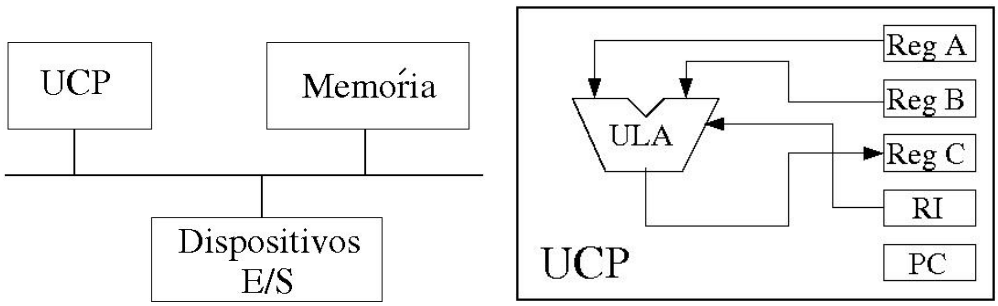


Figura 1: a) Componentes básicos de um computador e (b) elementos básicos da UCP.

1.1.1 Unidade Central de Processamento - UCP

A figura 1b mostra uma UCP em uma configuração muito simples. De forma geral, a UCP pode ser dividida em duas partes, a unidade lógica e aritmética (ULA) e a unidade de controle (UC). A ULA é capaz de desempenhar dois tipos de operações: operações aritméticas, como somas e subtrações, e comparações, como *igual a* ou *maior que*. A UC orquestra todo o resto. Seu trabalho é ler instruções e dados da memória ou dos dispositivos de entrada, decodificar as instruções, alimentar a ULA com as entradas corretas de acordo com as instruções e enviar os resultados de volta à memória ou aos dispositivos de saída. Um componente chave dos sistema de controle é um contador de programa (ou PC = program counter) que mantém o endereço da instrução corrente e que, tipicamente, é incrementado cada vez que uma instrução é executada, a não ser que a própria instrução

corrente indique onde se encontra a próxima instrução, permitindo assim que um conjuntos de instruções seja repetido várias vezes. Desde a década de 1980, a ULA e a UC são inseridas em um único circuito integrado: o microprocessador.

Há vários fabricantes e modelos de microprocessadores, como o Pentium da Intel, o Athlon da AMD e o PowerPC da IBM. Cada microprocessador possui um conjunto de instruções finito, que são executadas a uma determinada frequência. As frequências comuns atualmente giram entre 1 e 3 GHz (Giga Hertz). O microprocessador apenas busca a próxima instrução (ou dados) na memória e a executa, em um ciclo infinito (ou até desligarem o computador). A figura 1b mostra a ULA, recebendo informações dos registradores[1] A e B e colocando o resultado no registrador C. O registrador de instruções (RI) define a operação a ser executada. Esses registradores fazem parte da unidade de controle. A UC é capaz de configurar seus recursos (como registradores e a ULA) para executar cada instrução.

1.1.2 Memória

Conceitualmente, a memória do computador pode ser vista como uma lista de células. Cada célula tem um **endereço** numerado sequencialmente que pode armazenar uma quantidade fixa e pequena de informação. Essa informação pode ser ou uma **instrução**, que diz ao computador o que fazer, ou **dados**, a informação que o computador deve processar utilizando as instruções.

A memória pode ser classificada em 2 grupos, as memórias voláteis e não voláteis (ou permanentes). As memórias voláteis (memórias do tipo RAM - Random Access Memory) precisam de energia para manter seu conteúdo (ou seja, só funcionam quando o computador está ligado). Quando desligamos o computador, as informações importantes são armazenadas nos dispositivos de memória não voláteis (como o disco rígido ou HD - Hard Drive). Os dispositivos de memória volátil são mais caros e de menor capacidade, porém são muito mais rápidos, tornando possível ao computador realizar o processamento de forma mais eficiente. Tipicamente, um computador pessoal hoje tem 1GB a 4GB de RAM e 80GB a 250GB de HD.

1.1.3 Dispositivos de Entrada e Saída - E/S

Os dispositivos de E/S definem como o computador recebe informação do mundo exterior e como ele devolve informação para o mundo exterior. Teclados, mouses, *scanners*, microfones e câmeras são dispositivos comuns de entrada enquanto monitores e impressoras são dispositivos comuns de saída. Discos rígidos e placas de rede, que permitem conexões entre computadores, podem atuar como dispositivos tanto de entrada quanto de saída.

1.2 Linguagem de programação

Vimos que o processador executa uma instrução por vez, e que as instruções são bastante simples. Um programa pode ser definido como uma sequência de instruções, que ficam armazenadas na memória. Na prática, é muito difícil trabalhar diretamente com instruções da máquina. Durante o curso você aprenderá como solucionar problemas computacionais usando uma linguagem de alto nível como C. Um programa escrito em uma linguagem de alto nível é mais fácil de escrever e entender, porém não pode ser executado diretamente pelo microprocessador. É necessário converter o programa em linguagem de alto nível (conhecido como programa fonte) para um programa em linguagem de máquina (ou linguagem de baixo nível), que possa ser executado pelo computador. Essa conversão é realizada por um programa chamado de **compilador**.

E como instruções e dados podem ser utilizados para produzir algo útil? Imagine uma cozinha, contendo armários com diversos ingredientes, utensílios de cozinha, um forno e... um padeiro. O padeiro segue uma receita para fazer um bolo, a partir dos ingredientes e com o auxílio do forno e dos utensílios. Cada receita produz um tipo de bolo diferente, ou seja, mudando a receita, muda-se o bolo.

[1] Registradores são elementos de memória utilizados dentro da UCP

No computador-cozinha, os ingredientes constituem a entrada, os recipientes que o padeiro utiliza para armazenar os ingredientes (e talvez misturá-los) enquanto prepara o bolo são as variáveis e o bolo é a saída desejada. A receita constitui um algoritmo. Este conjunto forma o software. Já os utensílios, o forno e o próprio padeiro constituem as ferramentas necessárias no processo: o hardware.

Uma mesma receita pode ser escrita de várias formas diferentes. O algoritmo é uma entidade abstrata que define uma idéia. Sua realização, na forma de uma receita escrita, é semelhante a um programa de computador: um conjunto de instruções que, seguidas corretamente e ordenadamente, produzem o resultado desejado a partir das informações fornecidas.

Você já viu alguns algoritmos reais antes de chegar na universidade, como o algoritmo de divisão de inteiros. Mas como ensinar um computador a dividir, quando a linguagem disponível é muito limitada, por exemplo, quando só podemos realizar somas e subtrações?

Uma possível solução para se dividir um inteiro a pelo inteiro b usando apenas somas e subtrações é a seguinte:

```
1) Carregue os valores a e b nos registradores (ou posições de memória) RA e RB;
2) Carregue o valor zero em RC;
3) enquanto RA > RB repita:
  3.1) Incremente o valor em RC: RC = RC + 1;
  3.2) Subtraia o valor b de RA: RA = RA - RB;
4) imprima o valor em RC;
5) fim.
```

Por exemplo, seja $a = 12$ e $b = 5$. O resultado em RC recebe inicialmente o valor zero. Como 12 é maior que 5, o resultado é incrementado ($RC = 1$), e o valor do dividendo é atualizado ($RA = 12 - 5 = 7$). Como a condição $RA > RB$ ainda é verdadeira, as instruções 3.1 e 3.2 são repetidas mais uma vez, resultando em $RC = 2$ e $RA = 2$. Nesse instante, a condição $RA > RB$ se torna falsa, e então a instrução 4 é executada, sendo o valor 2 impresso como resultado da divisão 12/5.

Essa é uma maneira simples de se calcular a divisão entre números inteiros, e que embora seja apropriada para ser implementada em computadores, não é muito adequada para uso humano. Esse talvez seja o maior desafio desse curso, aprender a resolver problemas de forma computacional, que não são necessariamente ”naturais”para nós. Outro problema é descrever o problema em uma forma que o computador possa executar.

2 Um Primeiro Programa em C

Ronaldo F. Hashimoto, Carlos H. Morimoto e José A. R. Soares

O objetivo dessa aula é introduzir você à linguagem C em ambiente Linux, primeiramente mostrando a sua estrutura, e a seguir com um exemplo simples. Antes de iniciar essa aula, é desejável que você disponha de um editor de texto para escrever o programa, e verifique também a existência do compilador gcc em seu sistema.

Ao final dessa aula você deve ser capaz de:

- Descrever o ambiente em que os seus programas serão desenvolvidos.
- Escrever um programa em C que recebe dados do teclado e imprime dados na tela, usando as funções `scanf` e `printf`.
- Compilar um programa em C usando o gcc.
- Executar o programa após a compilação.

2.1 Sobre Linux

Todo programa roda em um ambiente definido pelo conjunto de hardware e software a sua disposição. Como mais de 90% dos computadores pessoais (PCs) no mundo usam algum sistema operacional da Microsoft, você provavelmente não sabe o que é Linux, e muito menos ainda sabe por que esses professores de Computação teimam em insistir que há alguma vantagem em usar Linux em seu PC, quando você está muito satisfeito (ou confortável) com o sistema que você possui agora.

O Linux começou a ser desenvolvido em 1991 por Linus Torvalds e é baseado no sistema operacional Unix. Esse projeto pode ser considerado hoje como um dos melhores exemplos de sucesso no desenvolvimento de software aberto (e que é grátis!). A maior aplicação do Linux se encontra em servidores (essas máquinas que mantém a Internet no ar) e por isso muitas companhias já apoiam esse sistema, como a Dell, HP e a IBM, entre outras. Como exemplo de usuário podemos citar a Google, que (estima-se) possui cerca de 450.000 servidores rodando Linux. Além de servidores, o Linux é utilizado também em supercomputadores, em plataformas de jogos como o PlayStation 2 e 3, em telefones celulares, e muitos outros sistemas computacionais. No entanto, apenas cerca de 1% dos *desktops* rodam Linux.

Para aprender a programar em C você não precisa instalar Linux em seu computador, pois há várias alternativas de ferramentas que você pode usar para desenvolver seus programas no ambiente Windows que seu professor pode lhe indicar. Porém, essa é uma excelente oportunidade de conhecer o Linux, o que pode-lhe trazer uma grande vantagem profissional no futuro (assim como, por exemplo, talvez seja importante aprender inglês e chinês). Um exemplo de projeto de grande porte que pode impulsionar ainda mais o uso de Linux é o projeto OLPC (*one laptop per child*), também conhecido como laptop de 100 dólares, que tem o potencial de atingir milhões de crianças em todo o mundo.

2.2 Interfaces Gráficas x Linha de Comando

Você provavelmente já está familiarizado com interfaces gráficas, essas que apresentam janelas, menus, ícones e outros componentes gráficos que você pode clicar, um cursor que você controla com o mouse, etc. Essas interfaces foram desenvolvidas na década de 1990, sendo que na década de 1980 as melhores interfaces eram do tipo `linha de comando`. Nesse tipo de interface, o monitor, em geral verde, era capaz de apresentar apenas texto. Dessa forma o usuário precisava digitar o nome do comando a ser executado pelo computador.

Atualmente, no Linux, você tem aplicações do tipo `Terminal`, que criam uma janela no ambiente gráfico onde você pode entrar com comandos usando o teclado [2]. É no `Terminal`, que você vai compilar e executar o seu

[2] Veja em nossa página como fazer isso no Windows, usando por exemplo uma janela do CYGWIN.

programa.

2.3 Sistema de Arquivos

As informações que você possui no computador são armazenadas nos dispositivos de memória não volátil (disco rígido ou HD) na forma de arquivos. De forma geral, podemos definir 3 tipos de arquivos: diretórios, dados e aplicativos. Diretórios são arquivos que contém outros arquivos e permitem que você organize todas as informações em seu disco rígido. Os outros tipos de arquivos contem informações. A diferença básica entre eles é que os aplicativos podem ser executados pelo computador, enquanto os dados (as vezes chamados de documentos) são utilizados como entrada e/ou saída dos aplicativos.

O compilador gcc, por exemplo, é um aplicativo [3] que recebe como entrada um arquivo fonte, e gera um arquivo executável (um outro aplicativo). O editor de texto é um exemplo de outro tipo de aplicativo, que não necessariamente precisa receber um arquivo de entrada, e pode gerar arquivos de dados na saída. Digamos que você use um editor de texto para escrever um programa em C, e salva esse programa no arquivo "exemplo.c". Embora esse arquivo contenha um programa, ele não pode ser executado enquanto não for traduzido para linguagem de máquina pelo compilador gcc.

Para esse curso, recomendamos que você sempre rode o gcc com as seguintes opções: "-Wall -ansi -O2 -pedantic". Essas opções garantem que o gcc vai lhe fornecer todas os avisos que ele é capaz de gerar para prevenir você contra possíveis falhas no seu programa. Assim, para compilar o arquivo "exemplo.c", podemos utilizar o seguinte comando na janela Terminal:

```
gcc -Wall -ansi -O2 -pedantic exemplo.c -o exemplo
```

A opção "-o" indica o nome do arquivo de saída, no caso, apenas "exemplo", sem nenhuma extensão.

2.4 O Esqueleto de um Programa em C

Finalmente, vamos ver qual o conteúdo de um arquivo com um programa em C. Para que você consiga compilar o seu programa em C sem problemas utilizando o gcc, todos os seus programas devem possuir o seguinte esqueleto:

```
1 #include <stdio.h>
2
3 int main()
4 {
5   /* declaração de variáveis */
6
7   /* lista de comandos */
8
9   return 0;
10 }
```

Por enquanto considere esse esqueleto como uma "receita de bolo", ou seja, todo programa em C deve conter os comandos das linhas $1, 3, 4, 9$ e 10.

2.5 Exemplo de um Programa

Para entendermos melhor, considere o programa em C apresentado na Fig. 2. Esse programa faz uma pergunta ao usuário (quantos anos você tem?), espera que o usuário entre com uma resposta numérica através do teclado,

[3] Muitas vezes chamamos aplicativos de programas, mas isso seria confuso em nosso contexto já que os programas que você vai escrever em C não podem ser executados antes de compilados.

e finaliza com um comentário sobre a idade que depende da resposta. Lembre que em C, assim como nos microprocessadores, as instruções são executadas sequencialmente, uma de cada vez.

```
1  # include <stdio.h>
2
3  int main() {
4
5    /* Primeiro programa em C */
6
7    /* declarações: todas as variáveis utilizadas precisam ser declaradas */
8
9    int idade;
10
11   /* início do programa */
12
13   printf ("Quantos anos voce tem?: ");
14   scanf ("%d", &idade);
15
16   printf ("%d? Puxa, voce parece que tem so %d anos!\n", idade, idade * 2);
17
18   /* fim do programa */
19
20   return 0;
21 }
```

Figura 2: Primeiro Programa

2.5.1 Comentários

Primeiramente, os textos entre os símbolos $/*$ e $*/$ (linhas $5, 7, 11$ e 18) são *comentários*. Comentários não interferem no programa, mas auxiliam os programadores a entender e documentar o código.

2.5.2 Declaração de Variáveis

Todo programa precisa de espaço na memória para poder trabalhar e as declarações reservam o espaço necessário para isso. Na linha 9, temos uma declaração de uma variável de nome `idade`. Esta variável guarda números de tipo **int** (inteiro). Em C, todas as variáveis utilizadas precisam ser declaradas no início de cada bloco de comandos. A forma de declaração de variáveis é:

int <nome_da_variavel>;

2.5.3 Funções de Leitura e Impressão

Todos os seus programas devem se comunicar com o usuário através de funções de impressão (na tela) e de leitura (pelo teclado). Basicamente, nos nossos primeiros programas, o usuário fornece números inteiros para o programa através da leitura pelo teclado (função `scanf`); enquanto que o programa fornece ao usuário os resultados via impressão de mensagens na tela (função `printf`). No nosso exemplo, a função de impressão na tela está sendo utilizada nas linhas 13 e 16; enquanto que a função de leitura pelo teclado está sendo utilizada na linha 14.

2.5.4 Função de Impressão na Tela

Basicamente, a função `printf` imprime todos os caracteres que estão entre aspas. Assim, o `printf` da linha 13 imprime a mensagem (sem as aspas) `"Quantos anos voce tem?: "`. Note que o espaço em branco no final da mensagem também é impresso!

Agora observe o `printf` da linha 16. Este `printf` tem duas diferenças com relação ao `printf` da linha 13. A primeira diferença é que dentro da mensagem do `printf` da linha 16 (caracteres que estão entre aspas) podemos encontrar duas sequências de caracteres: `"%d"` e `"\n"`. Além disso, depois da mensagem, temos duas expressões aritméticas envolvendo a variável `idade` separadas por vírgulas: (a) `"idade"` (seria como a expressão aritmética `"idade * 1"`) ; e (b) a expressão aritmética `"idade * 2"`.

O `printf` da linha 16 imprime na tela todos os caracteres que estão entre aspas, com exceção da sequência de caracteres `"%d"` e `"\n"`.

Para cada sequência de caracteres `"%d"`, a função `printf` imprime na tela um número inteiro que é resultado das expressões aritméticas contidas no `printf` separadas por vírgula. Assim, o primeiro `"%d"` imprime na tela o conteúdo da variável `"idade"` e o segundo `"%d"` imprime na tela o resultado da expressão `"idade * 2"` (uma vez que a expressão `"idade"` vem antes da expressão `"idade * 2"` no `printf` da linha 16).

A sequência de caracteres `"\n"`, indica à função `printf` para "pular de linha", isto é, faz com que o cursor da tela vá para a próxima linha. No `printf` da linha 16, como a sequência está no final da mensagem, isto significa que depois de imprimir a mesma na tela, o cursor irá para a próxima linha.

2.5.5 Função de Leitura pelo Teclado

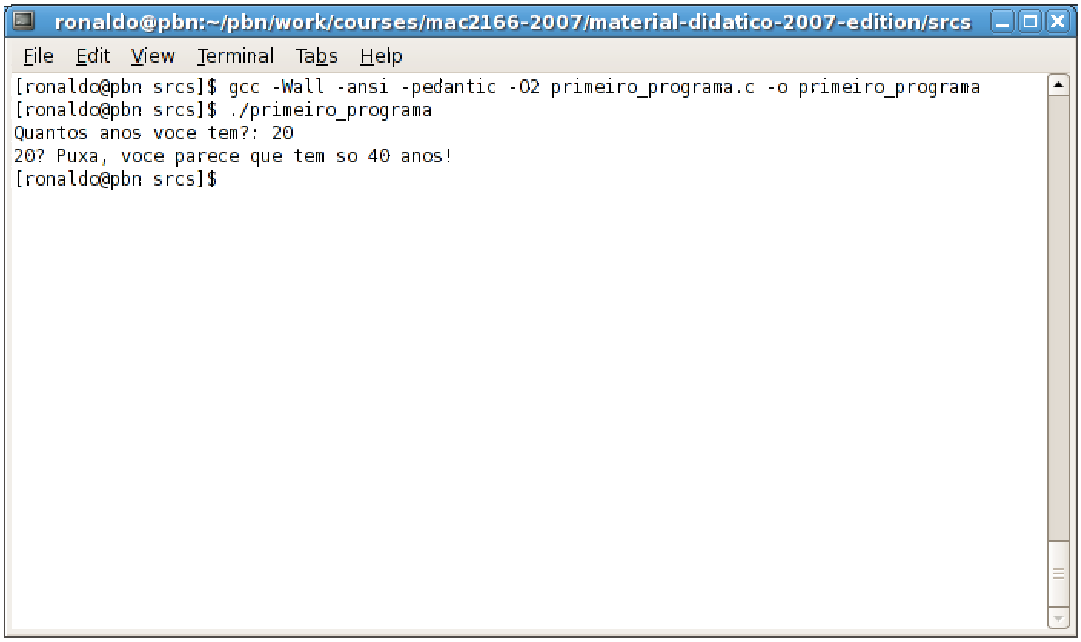
Para ler um número inteiro pelo teclado, você deve usar a função `scanf` da seguninte forma:

```
scanf ("%d", &<nome_da_variavel>);
```

o `scanf` irá esperar o usuário digitar um número inteiro pelo teclado e, após o usuário digitar a tecla `<ENTER>`, armazenará o número digitado na variável `<nome_da_variavel>`. Um exemplo está na linha 14 do primeiro programa: o número digitado irá ser armazenado na variável `idade`. Observe que no `scanf` deve-se colocar o caractere `"&"` antes do nome da variável.

2.5.6 Retornando ao nosso Exemplo

Executando o programa, temos:



Observe que:

1. O número `"20"` que aparece depois da mensagem

   ```
   "Quantos anos voce tem?:  "
   ```

   foi digitado pelo usuário e lido pela função `scanf`.

2. Este número `"20"` aparece ao lado da mensagem, pois o `printf` que a imprime na tela não tem a sequência de caracteres `\n` no final; caso contrário, o número `"20"` seria digitado na próxima linha.

3. Uma vez que o usuário, depois de digitar o número 20, deve dar um `<ENTER>`, o cursor automaticamente irá para a próxima linha; observe que a mensagem

   ```
   "20?  Puxa voce parece que tem so 40 anos!"
   ```

   aparece na próxima linha.

4. No segundo `printf`, os números `"20"` e `"40"` (resultados das expressões aritméticas `"idade"` e `"idade * 2"`) são colocados no lugar da primeira e segunda sequência de caracteres `"%d"`, respectivamente.

2.5.7 Impressão de `"%d"` e `"\n"`

Para imprimir na tela a sequência de caracteres `"%d"`, você deve usar

```
printf ("%%d");
```

e para imprimir `"\n"`, você deve usar

```
printf ("\\n");
```

2.6 Mais detalhes sobre o esqueleto

A linguagem C é uma linguagem de alto nível criada por Brian Kernighan e Dennis Ritchie no início da década de 1970 nos laboratórios da AT&T Bell, e suas origens estão relacionadas ao desenvolvimento do sistema operacional Unix. O C é também uma linguagem estruturada, que permite que um programa complexo seja facilmente decomposto em programas mais simples, definindo assim módulos.

Cada módulo básico é chamado de função, e cada função precisa ter um nome (ou identificador) bem definido e diferente dos demais. No caso, a função de nome `main` é necessária em todos os programas pois define o início da execução do programa. A função `main` foi definida no esqueleto como uma função `int` (ou seja, inteira), e por isso precisa devolver um valor inteiro. Daí a necessidade do comando `return 0`, apenas por consistência, já que o zero não é realmente utilizado. Toda função em C recebe também parâmetros. Por exemplo, uma função seno deve receber como parâmetro um ângulo. A lista de parâmetros em C é declarada entre parênteses depois do nome e, no caso da função `main`, ela recebe zero parâmetros pois não há nada entre os parênteses. As chaves definem o início e fim de um bloco de instruções.

Embora os comandos da linguagem C sejam bem poderosos, eles são limitados. Mas com a maturidade de uma linguagem, vários programadores desenvolveram funções auxiliares que facilitam a programação e podem ser compartilhadas com outros programadores. Para utilizar essas funções, basta que você especifique onde encontrá-las através das linhas de `include`. No caso, o pacote `stdio.h` contém as funções necessárias para ler caracteres do teclado e imprimir caracteres no monitor, ou seja, contém as funções `scanf` e `printf`.

3 Fundamentos

Ronaldo F. Hashimoto, Carlos H. Morimoto e José A. R. Soares

Essa aula introduz vários fundamentos necessários para compreender a linguagem C e o funcionamento de seus comandos. Ao final dessa aula você deverá saber:

- Declarar e utilizar variáveis.
- Descrever a precedência dos operadores e como isso afeta o resultado das expressões aritméticas.
- Utilizar expressões aritméticas e relacionais, e prever seus resultados para entradas conhecidas.
- Utilizar comandos de leitura, impressão e atribuição.

3.1 Declaração de Variáveis

A declaração de uma variável que guarda números inteiros em C de nome `<nome_da_variavel>` é feita da seguinte forma:

int `<nome_da_variavel>`;

Exemplo: declaração de uma variável inteira `"soma"`

int `soma`;

Se você quiser declarar várias variáveis, é possível fazer da seguinte forma:

int `<nome_da_variavel_1>`, `<nome_da_variavel_2>`, `<nome_da_variavel_3>`,..., `<nome_da_variavel_n>`;

Exemplo: declaração de duas variáveis inteiras `"num"` e `"soma"`.

int `num`, `soma`;

3.2 Expressão Aritmética

Expressões aritméticas são expressões matemáticas envolvendo números inteiros, variáveis inteiras, e os operadores `"+"` (soma), `"-"` (subtração), `"/"` (quociente de uma divisão inteira), `"%"` (resto de uma divisão inteira) e `"*"` (multiplicação).

Exemplos:

- `num1` + `num2` $*$ 3
- `num` + 3 / 2
- `num` $*$ 3 + 2

Operador Aritmético	Associatividade
*, /, %	da esquerda para a direita
+, -	da esquerda para a direita

Tabela 1: Precedência dos Operadores Aritméticos.

3.2.1 Precedência de Operadores

Qual seria o resultado da expressão: $2 + 3 * 4$? Sua resposta provavelmente seria 14, pois é o resultado de $2 + (3 * 4)$, mas porque não 20, resultado de $(2 + 3) * 4$? A resposta está na prioridade com que as operações são realizadas, ou precedência dos operadores. A operação `"*"` tem maior precedência que a operação `"+"`, e portanto é feita primeiro.

A Tabela 1 mostra a precedência dos operadores em C. Na dúvida, ou até para deixar mais claro e fácil de entender, use parênteses. Além de números as expressões podem conter o nome de variáveis, como na soma `"num1 + num2"`.

Um outro fator importante é o tipo dos valores utilizados pelos operadores, no caso, estamos trabalhando apenas com o tipo inteiro (**int**). Isso é muito importante para entender o resultado de algumas expressões. Por exemplo, usando agora o compilador, faça um programa que imprima o valor da expressão `(3 / 4 * 100)`. O resultado é **zero**. Por quê?

Como a precedência de `/` e `*` são iguais, a tabela diz também que esses operadores são calculados da esquerda para a direita, ou seja, o resultado de `3/4` é multiplicado por `100`, e o resultado final esperado seria `75`. Porém, o resultado do seu programa deve ter sido **zero**. Por que isso?

Como todas as operações são inteiras, o resultado de `3/4` é **zero** (e não `0.75`, que é um número real). Sendo assim, o resultado de `9/2` é `4`, `9/3` é `3`, `9/4` é `2`, e assim por diante. A parte fracionária é simplesmente eliminada (ou truncada ao invés de ser aproximada para um valor inteiro mais próximo), ou seja, mesmo o resultado de `99999/100000` é **zero**.

Considere as variáveis inteiras `x` $= 2$ e `y` $= 3$. Verifique o valor das seguintes expressões:

Expressão	Valor
`x / y`	0
`y / x`	1
`y / x` $*$ 10	10
`x` + `y` $*$ 4	14
(`x` + `y`) $*$ 4	20

3.3 Expressão Relacional

Várias instruções depedem do resultado de comparações (ou condições) do tipo `num1` $>$ `num2` (`num1` é maior que `num2`). O resultado de uma condição é **verdadeiro** ou **falso**.

Expressões relacionais são expressões que envolvem comparações simples envolvendo operadores relacionais `"<"` (menor), `">"` (maior), `"<="` (menor ou igual), `">="` (maior ou igual), `"!="` (diferente), `"=="` (igual).

Uma comparação simples só pode ser feita entre pares de expressões aritméticas da forma:

<expr_aritmética_01> <oper_relacional> <expr_aritmética_02>

onde <expr_aritmética_01> e <expr_aritmética_02> são expressões aritméticas e <oper_relacional> é um operador relacional.

No decorrer do curso iremos aprender como fazer comparações mais complexas utilizando operadores lógicos. Vamos deixar este tópico para ser discutido mais adiante.

3.4 Leitura pelo Teclado

A leitura de um número inteiro pelo teclado (fornecido pelo usuário) é feita usando a função `scanf` da seguinte forma:

```
scanf ("%d", &<nome_da_variavel>);
```

Exemplo: `scanf ("%d",&idade);`

É possível também ler dois ou mais números. Por exemplo,

```
scanf ("%d %d %d", &<nome_da_variavel_01>, &<nome_da_variavel_02>, &<nome_da_variavel_03>);
```

lê três números inteiros do teclado armazenando-os na variáveis `<nome_da_variavel_01>`, `<nome_da_variavel_02>` e `<nome_da_variavel_03>`. Observe que o `scanf` tem três sequências de `"%d"` e tem um `"&"` antes de cada variável.

Se você tem dúvida de como funciona isto, faça um programa simples que leia dois inteiros via teclado (com somente um `scanf`) e imprima sua soma.

3.5 Impressão na Tela

A impressão de uma mensagem na tela é feita usando a função `printf`. A mensagem deve ser colocada entre aspas da seguinte forma:

```
printf ("<mensagem>");
```

Basicamente, a função `printf` imprime todos os caracteres que estão entre aspas, com exceção da sequência de caracteres `"%d"` e `"\n"`.

Considere o exemplo:

```
printf ("Os numeros lidos foram %d e %d\n", num1, num2);
```

Para cada sequência de caracteres `"%d"`, a função `printf` imprime na tela um número inteiro que é resultado das expressões aritméticas contidas no `printf` separadas por vírgula. Assim, o primeiro `"%d"` imprime na tela o conteúdo da variável `"num1"` e segundo `"%d"` imprime na tela o resultado da expressão `"num2"` (uma vez que a expressão com a variável `"num1"` vem antes da expressão com a variável `"num2"` no `printf` do exemplo acima.

Se você tem dúvidas, compile e execute o programa abaixo:

```c
#include <stdio.h>
#include <stdlib.h>

int main () {

  /* declaracoes */

  int num1, num2;

  /* programa */

  printf("Entre com dois numeros inteiros: ");
  scanf("%d %d", &num1, &num2);

  printf ("Os numeros lidos foram %d e %d\n", num1, num2);

  /* fim do programa */

  return 0;
}
```

3.6 Atribuição

Suponha que você queira guardar a soma dos dois números lidos do programa anterior em uma outra variável de nome `soma`. Para isso, devemos usar uma atribuição de variável. A atribuição de uma variável é uma operação que armazena o resultado de uma expressão aritmética (`expr_arimética`) em uma variável (`nome_da_variável`) da seguinte forma:

```
nome_da_variável = expr_arimética;
```

Em uma atribuição, a variável (**SEMPRE UMA E UMA ÚNICA VARIÁVEL**) do lado **esquerdo** do símbolo `=` recebe o valor da expressão aritmética do lado **direito**.

Exemplos:

- `soma = num1 + num2;`
- `z = x / y;`
- `z = y / x;`
- `z = y / x * 10;`
- `x = x + y * 4;`
- `y = (x + y) * 4;`

3.6.1 Atribuição e Comparação

Note a diferença entre o operador de atribuição = e o operador relacional ==. Observe os comandos:

1. `a=b`
2. `a==b`

O primeiro armazena o conteúdo da variável `b` na variável `a`. O segundo, com significado bem diferente, compara se o conteúdo da variável `a` é igual ao conteúdo da variável `b`.

3.6.2 Um Programa para Testar

Se você tem dúvidas, compile e execute o programa abaixo:

```
1 # include <stdio.h>
2 # include <stdlib.h>
3
4 int main () {
5
6   /* declaracoes */
7
8   int num1, num2, soma;
9
10  /* programa */
11
12  printf("Entre com dois numeros inteiros: ");
13  scanf("%d %d", &num1, &num2);
14
15  printf ("Os numeros lidos foram %d e %d\n", num1, num2);
16
17  soma = num1 + num2;
18
19  printf("O resultado da soma de %d com %d eh igual a %d\n", num1, num2, soma);
20
21  /* fim do programa */
22
23  return 0;
24 }
```

3.7 Dicas

- Preste MUITA atenção ao digitar o seu programa. É muito fácil "esquecer" um ponto-e-vírgula, ou esquecer de fechar chaves e parênteses.
- Leia com cuidado as mensagens do compilador. A maioria das mensagens de *warning* são causadas por erros de lógica ou digitação. Por exemplo, se você digitar "=" ao invés de "==" em uma expressão relacional, o compilador gerará um *warning*.
- Na linguagem C, caracteres minúsculos e maiúsculos são diferenciados. Assim, as variáveis `num1`, `Num1`, `NUm1`, e `NUM1` são todas diferentes.
- Procure utilizar nomes significativos para variáveis. Ao invés de `a`, `b` e `c`, você pode utilizar algo como `idade`, `altura` e `peso`.
- Você não pode utilizar palavras reservadas como `int`, `if`, `for`, `while`, etc., bem como nome de suas variáveis.

4 Comando de Repetição: while

Ronaldo F. Hashimoto e Carlos H. Morimoto

Essa aula introduz o comando `while`, que permite repetir instruções enquanto uma condição for verdadeira. Para utilizar o comando corretamente, você precisa se lembrar de inicializar as variáveis de controle antes do comando, certificar-se que a condição do `while` se mantem verdadeira pelo número correto de iterações, e por fim garantir que a condição se torne falsa para terminar o looping.

Ao final dessa aula você deverá saber:

- Utilizar comandos de repetição na resolução de problemas computacionais.
- Definir condições iniciais e de parada para o comando `while`.
- Simular o processamento do comando `while`.

4.1 Sintaxe

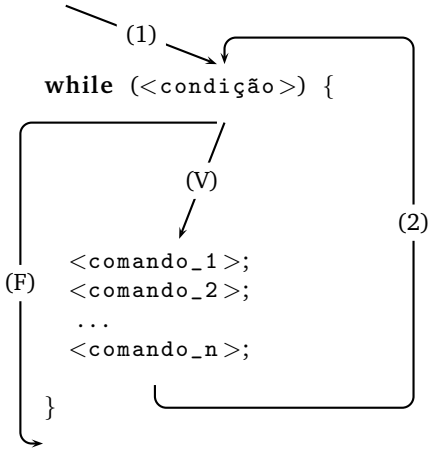
A sintaxe do comando de repetição do comando `while` pode ser vista ao lado.

A <condição> é uma expressão relacional que tem como resultado um valor **verdadeiro** ou **falso** (veja aula sobre **fundamentos**). A sequência de comandos <comando_1>, <comando_2>, ..., <comando_n> pode conter comandos de atribuição, impressão de mensagens na tela ou leitura de números inteiros pelo teclado, entre outros.

```
while (<condição>) {
  <comando_1>;
  <comando_2>;
  ...
  <comando_n>;
}
```

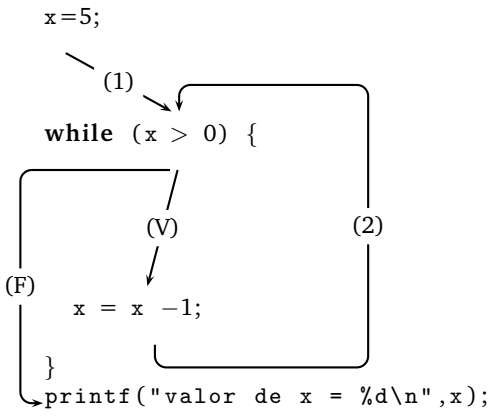
4.2 Descrição

Basicamente, este comando de repetição tem o significado: enquanto a <condição> for **verdadeira**, a sequência de comandos <comando_1>, <comando_2>, ..., <comando_n> é executada.



Vamos analisar o "fluxo" do programa usando o comando de repetição **while**. Primeiramente, quando a execução do programa chega no **while** (seta marcada com (1)) a <condição> é testada. Se "de cara" a <condição> é **falsa**, o fluxo do programa ignora a sequência de comandos e segue a seta marcada com (F). Agora, se a <condição> é **verdadeira**, então o fluxo do programa segue a seta marcada com (V) e executa a sequência de comandos dentro do **while**; após executado o último comando (<comando_n>), o fluxo do programa segue a seta marcada com (2) e volta a testar a <condição>. Se a <condição> é **verdadeira**, então o fluxo do programa segue a seta marcada com (V) repetindo a sequência de comandos dentro do **while**. Se <condição> é **falsa**, o fluxo do programa ignora a sequência de comandos e segue a seta marcada com (F).

Por exemplo, seja `x` uma variável inteira. O segmento de programa abaixo simplesmente subtrai 1 de `x`, 5 vezes (note que o comando `"x = x-1;"` é repetido 5 vezes).



O primeiro comando, a atribuição `x = 5;` (a variável `x` recebe o valor `5`), é executado antes do **while** (condição inicial da variável que controla o **while**). Depois o fluxo do programa segue a seta marcada com `(1)` e testa a condição `(x>0)` do **while**. Se ela é **verdadeira**, executa os comandos dentro do **while** (que nesse caso contém apenas a atribuição `x = x − 1;`) seguindo a seta marcada com `(V)`. Depois o fluxo do programa segue a seta marcada com `(2)` e a condição `(x>0)` é testada novamente (agora `x` tem um valor decrescido de um). Dessa forma, o comando `x = x − 1;` é executado enquanto a condição do **while** é **verdadeira**. Somente quando a condição for **falsa**, o **while** termina, seguindo a seta marcada com `(F)` e a instrução seguinte é executada (no caso, o `printf`).

NOTA: para que o seu programa termine, você precisa garantir que a <`condição`> do **while** seja alterada de alguma forma, ou seja, você precisa garantir que a condição de parada seja alcançada. Caso contrário, o programa entra em "looping infinito".

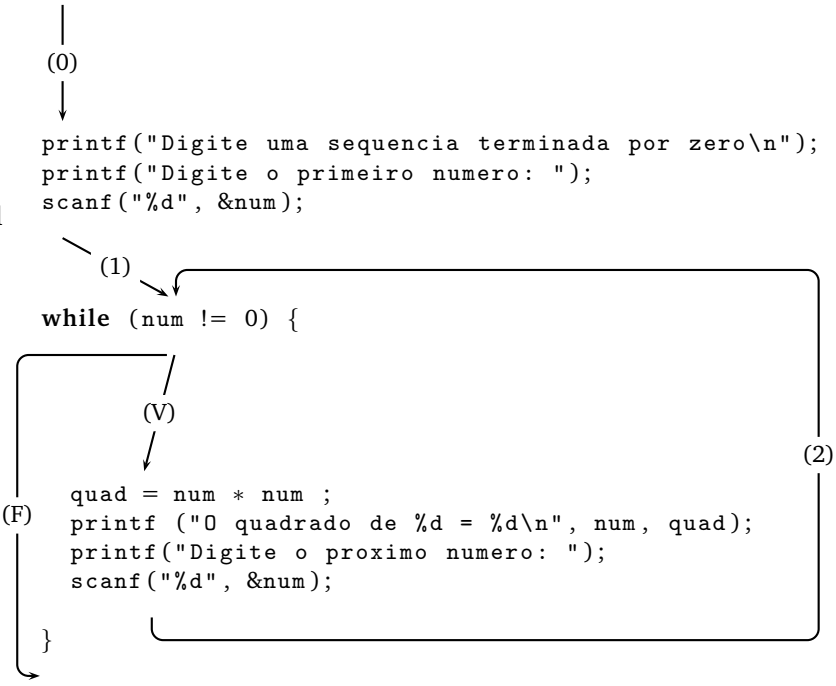
4.3 Exemplos Comentados

4.3.1 Exemplo 1

Dada uma sequência de números inteiros diferentes de zero, terminada por um zero, imprima o quadrado de cada número da sequência.

Solução:
Uma solução possível pode ser descrita de modo informal como:

1. imprima uma mensagem para o usuário saber o que fazer
2. leia pelo teclado o primeiro número da sequência na variável `num`
3. enquanto `num` for diferente de zero faça:
   (a) calcule `quadrado = num * num`
   (b) imprima na tela o valor de `quadrado`
   (c) leia pelo teclado o próximo número da sequência na variável `num`
4. fim



O funcionamento do programa pode ser entendido também pelo diagrama ao lado.

Em geral, é mais simples desenhar o diagrama e, quando você estiver certo de que ele funciona, sua "tradução" para a linguagem C é simples, basta copiar o esqueleto de um programa em C visto anteriormente, e preencher as lacunas. O programa em C ficaria:

```c
# include <stdio.h>
# include <stdlib.h>

int main () {
  /* declaracoes */
  int num; /* variavel utilizada para leitura da sequencia */
  int quad; /* variavel que armazena o quadrado de um numero */

  /* programa */
  printf("Digite uma sequencia terminada por zero\n");
  printf("Digite o primeiro numero: ");
  scanf("%d", &num);

  while (num != 0) {
    /* os simbolos '!=' significam diferente */
    quad = num * num ;
    printf ("O quadrado de %d = %d\n", num, quad);
    printf("Digite o proximo numero: ");
    scanf("%d", &num);
  }

  /* fim do programa */
  return 0;
}
```

4.3.2 Exemplo 2

Dada uma sequência de números inteiros diferentes de zero, terminada por zero, calcular a somatória dos números da sequência.

Solução:
Para melhor entender o problema, vamos ver um exemplo concreto de uma sequência numérica. Para a sequência:

$$2 \quad 3 \quad -4 \quad 5 \quad 0$$

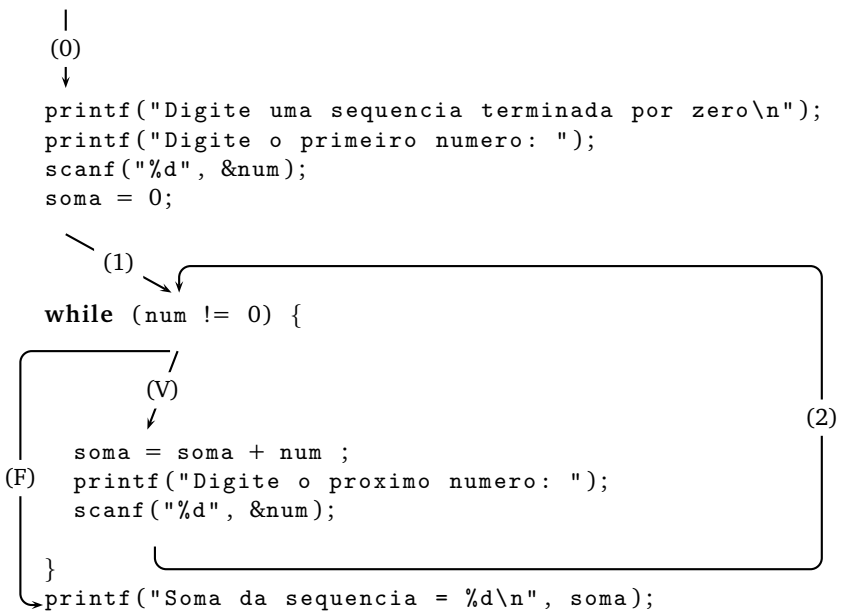
a saída de seu programa deve ser 6 (ou seja, $2 + 3 - 4 + 5$).

Uma forma possível para resolver esse problema é imaginar uma variável que armazena as somas parciais. Essa variável deve iniciar com o valor zero, e para cada número da sequência, ser somada com mais esse número, até o final da sequência. Assim, para o exemplo acima, o valor de soma torna-se 2 após processar o primeiro elemento da sequencia (soma-se o 2), 5 após o segundo (soma-se o 3), 1 após o terceiro (soma-se o 4), e assim até o final.

Uma solução possível pode ser descrita de modo informal como:

1. imprima uma mensagem para o usuário saber o que fazer
2. leia pelo teclado o primeiro número da sequência na variável num
3. inicialize uma variável soma com zero
4. enquanto num for diferente de zero faça:
   (a) acumule na variável soma o número lido
   (b) leia pelo teclado o próximo número da sequência na variável num
5. imprima na tela a soma final
6. fim

O funcionamento do programa pode ser entendido também pelo diagrama abaixo:



O programa completo ficaria:

```
1  # include <stdio.h>
2  # include <stdlib.h>
3  int main () {
4    /* declaracoes */
5    int num;  /* variavel utilizada para leitura da sequencia */
6    int soma; /* variavel que armazena a soma da sequencia */
7
8    /* programa */
9    printf("Digite uma sequencia terminada por zero\n");
10   printf("Digite o primeiro numero: ");
11   scanf("%d", &num);
12
13   while (num != 0) {
14       soma = soma + num ;
15       printf("Digite o proximo numero: ");
16       scanf("%d", &num);
17   }
18
19   printf("Soma da sequencia = %d\n", soma);
20
21   /* fim do programa */
22   return 0;
23 }
```

4.3.3 Exercícios Recomendados

1. (exercício 4 da lista) Dados números inteiros n e k, com $k >= 0$, determinar n^k (n elevado a k). Por exemplo, dados os números 3 e 4 o seu programa deve escrever o número 81.
2. (exercício 8 da lista) Dado um número inteiro $n >= 0$, calcular o fatorial de n (n!).

A solução para esses e outros exercícios você encontra na lista de exercícios em
`http://www.ime.usp.br/~macmulti/exercicios/inteiros/index.html`.

5 Comando de Seleção Simples e Composta

Ronaldo F. Hashimoto e Carlos H. Morimoto

Essa aula introduz o comando de seleção, que permite ao seu programa tomar decisões sobre o fluxo do processamento, ou seja, dependendo do estado ou condição de seu programa, esse comando permite selecionar que procedimento deve ser executado a seguir.

Ao final dessa aula você deverá saber:

- Utilizar o comando if.
- Utilizar o comando if-else
- Identificar situações onde é necessário utilizar o comando if-else e situações onde é suficiente utilizar o comando if.
- Simular o processamento dos comandos if e if-else em um programa.

5.1 Sintaxe

A sintaxe do comando de seleção é ilustrada na figura 3.

A <condição> é uma expressão relacional que tem como resultado um valor **verdadeiro** ou **falso** (veja a aula sobre **fundamentos**). A sequência de comandos dentro do **if** <comando_1>, <comando_2>, ..., <comando_n>, bem como a sequência de comandos dentro do **else** <outro−comando_1>, <outro−comando_2>, ..., <outro−comando_m>, podem ser comandos de atribuição, impressão de mensagens na tela, leitura de números inteiros pelo teclado ou eventualmente um outro comando de seleção.

Seleção Simples	Seleção Composta

if (<condição>) {
  <comando_1>;
  <comando_2>;
  ...
  <comando_n>;
}
	if (<condição>) {
	  <comando_1>;
	  <comando_2>;
	  ...
	  <comando_n>;
	} **else** {
	  <outro−comando_1>;
	  <outro−comando_2>;
	  ...
	  <outro−comando_m>;
	}

Figura 3: Sintaxe dos comandos de seleção simples e composta.

Observação: se dentro do **if** ou do **else** existir apenas um comando, não é necessário colocá-lo entre chaves. A figura 4 mostra a sintaxe do **if** nesse caso.

Seleção Simples	Seleção Composta
1 **if** (<condição>) 2 <comando>;	1 **if** (<condição>) 2 <comando>; 3 **else** 4 <outro_comando>;

Figura 4: Sintaxe dos comandos de seleção com comando único. Note que as chaves não são necessárias nesse caso.

5.2 Descrição

5.2.1 Seleção simples

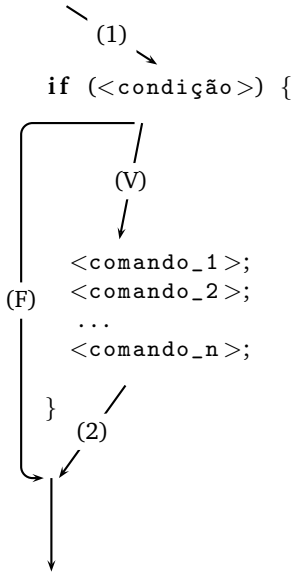


Figura 5: Fluxo do comando de seleção simples.

Basicamente, o comando de seleção simples tem o seguinte significado: se a <condição> for **verdadeira**, a sequência de comandos <comando_1>, <comando_2>, ..., <comando_n> é executada. Caso contrário, a sequência de comandos <comando_1>, <comando_2>, ..., <comando_n> não é executada.

A figura 5 ilustra a execução desse comando em um programa. Primeiramente a <condição> do **if** é testada (seta marcada com (1)). Se "de cara" a <condição> é **falsa**, o fluxo do programa ignora a sequência de comandos dentro do **if** e segue a seta marcada com (F). Agora, se a <condição> é **verdadeira**, então o fluxo do programa segue a seta marcada com (V) e executa a sequência de comandos dentro do **if**; executado o último comando (<comando_n>), o fluxo do programa segue a seta marcada com (2) e o programa segue adiante.

5.2.2 Seleção Composta

O comando de seleção composta tem o seguinte significado: se a <condição> for **verdadeira**, a sequência de comandos <comando_1>, <comando_2>, ..., <comando_n> é executada. Caso contrário a sequência executada é a sequência de comandos <outro_comando_1>, <outro_comando_2>, ..., <outro_comando_m>.

Vamos analisar o "fluxo" do comando de seleção composta **if**−**else** através da figura 6. Primeiramente, a execução do programa vem e testa a <condição> do **if** (seta marcada com (1)). Se a <condição> é **verdadeira**, o fluxo do programa segue a seta marcada com (V) e executa a sequência de comandos dentro do **if** e ignora a sequência de comandos dentro do **else** seguindo a seta marcada com (2) a instrução seguinte do comando **if**−**else** é executada. Agora, Se a <condição> do **if** é **falsa**, o fluxo do programa ignora a sequência de comandos dentro do **if** e segue a seta marcada com (F) e executa a sequência de comandos dentro do **else**. No final o fluxo segue a seta marcada com (3) executando a instrução seguinte ao comando **if**−**else**.

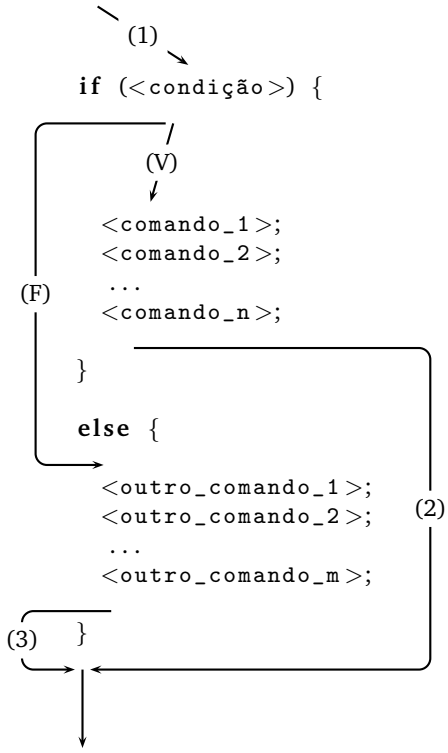


Figura 6: Fluxo do comando de seleção composta.

5.3 Exemplos Comentados

5.3.1 Exemplo 1

Vamos analisar o "fluxo" de um programa usando o comando de seleção simples **if** aplicado a um problema bastante simples:

Escreva um programa em C que leia umn número inteiro e devolve o seu valor absoluto.

Solução: Esse problema apenas ilustra a utilização do comando **if**. Sabemos que para calcular o módulo (valor absoluto) de um número inteiro basta trocar o seu sinal quando ele for negativo. Portanto, uma solução simples para esse problema seria:

1. imprima uma mensagem para o usuário saber o que fazer;
2. leia pelo teclado um número inteiro em num1;
3. se num1 for negativo, transforme-o para positivo;
4. imprima o resultado
5. fim

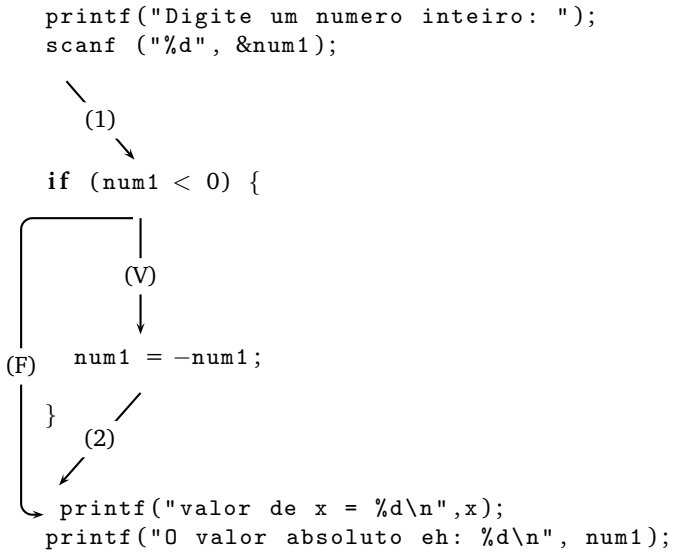


Figura 7: Cálculo do módulo de um número inteiro.

O funcionamento desse programa pode ser entendido pelo diagrama da figura 7. Esse trecho de programa lê do teclado um número inteiro antes do comando **if** e o armazena na variável num1. A condição do **if** é então testada e no caso de num1 ser menor que zero, o programa simplesmente troca o sinal de num1 e no final imprime o novo valor de num1. Observe que se o número digitado for positivo, o sinal de num1 não é trocado, e seu valor é impresso na tela.

A solução completa é dada abaixo:

```
1  # include <stdio.h>
2  # include <stdlib.h>
3
4  int main () {
5    int num1;
6    printf("Digite um numero inteiro: ");
7    scanf("%d ", &num1);
8
9    if (num1 < 0) {
10     num1 = −num1;
11   }
12
13   printf("O valor absoluto eh: %d\n", num1);
14   return 0;
15 }
```

5.3.2 Exemplo 2

Escreva um programa que leia 2 números inteiros e imprima o maior.

Solução:

A simplicidade desse programa permite observar o uso do comando if-else. Para isso, vamos utilizar 2 variáveis para armazenar os valores de entrada, e outra para armazenar o maior valor. Uma solução possível pode ser descrita de modo informal como:

1. imprima uma mensagem para o usuário saber o que fazer
2. leia pelo teclado dois números inteiros num1 e num2
3. se num1 maior que num2
   (a) maior = num1
4. senão
   (a) maior = num2
5. imprime o conteúdo da variável maior
6. fim

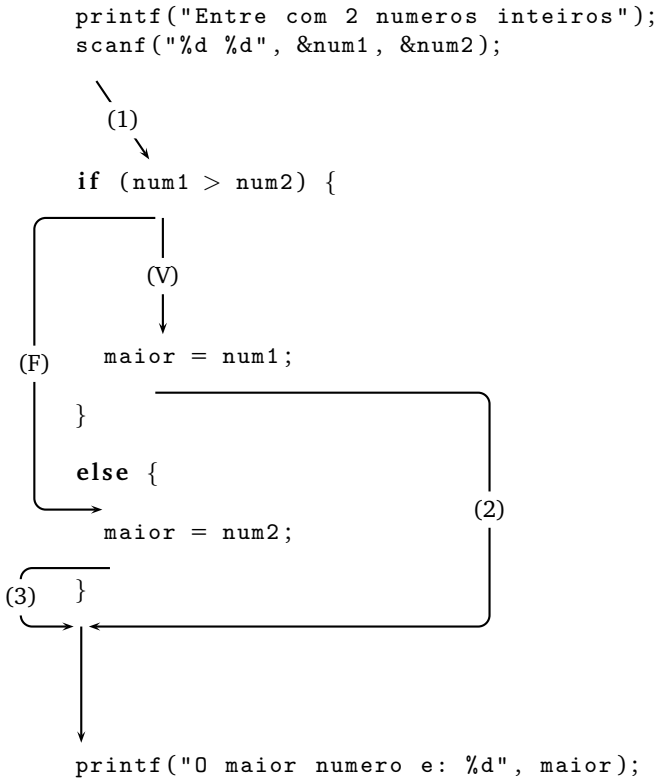


Figura 8: Lê 2 números e imprime o maior.

O funcionamento do programa pode ser entendido também pelo diagrama da figura 8.

A solução completa é dada abaixo:

```
1  # include <stdio.h>
2  # include <stdlib.h>
3
4  int main () {
5    int num1, num2, maior;
6    printf("Entre com 2 numeros inteiros");
7    scanf("%d %d", &num1, &num2);
8
9    if (num1 > num2) {
10     maior = num1;
11   }
12   else {
13     maior = num2;
14   }
15
16   printf("O maior numero e: %d", maior);
17   return 0;
18 }
```

Observe que basta comparar `num1` com `num2` para sabermos qual o maior.

5.3.3 Exemplo 3

Dados um número inteiro `n`>0 e uma sequência com `n` números inteiros, determinar a soma dos inteiros positivos da sequência. Por exemplo, para `n`$=7$ e para a sequência com `n`$=7$ números inteiros

$$6 \quad -2 \quad 7 \quad 0 \quad -5 \quad 84$$

o seu programa deve escrever o número 25.

Solução:

Uma forma possível para resolver esse problema é imaginar uma variável que armazena as somas parciais. Essa variável deve iniciar com o valor zero, e para cada número da sequência, se o número lido é positivo, ser somada com mais esse número, até o final da sequência. Assim, para o exemplo acima, o valor de soma torna-se 6 após processar o primeiro elemento da sequencia (soma-se o 6), 13 após o terceiro (soma-se o 7), 21 após o sexto (soma-se o 8), e assim até o final.

Uma solução possível pode ser descrita de modo informal como:

1. imprima uma mensagem para o usuário saber o que fazer
2. leia pelo teclado a quantidade de números da sequencia na variável n
3. inicialize uma variável soma com zero
4. enquanto n maior que zero faça:
   (a) leia pelo teclado o número da sequência na variável num
   (b) se num maior que zero
      i. acumule na variável soma o número lido
   (c) diminua 1 da variável n
5. imprima na tela a soma final
6. fim

Uma solução do problema escrito em C:

```
1  # include <stdio.h>
2  # include <stdlib.h>
3
4  int main () {
5    /* declaracoes */
6    int num;  /* variavel utilizada para leitura da sequencia */
7    int soma; /* variavel que armazena a soma da sequencia */
8
9    /* programa */
10   printf("Digite a quantidade de numeros da sequencia: ");
11   scanf("%d", &n);
12
13   while (n > 0) {
14
15     printf("Digite um numero: ");
16     scanf("%d", &num);
17
18     if (num > 0)
19       soma = soma + num ;
20
21
```

```
22      }
23
24      printf("Soma dos numeros positivos da sequencia = %d\n", soma);
25      /* fim do programa */
26      return 0;
27    }
```

Observação: note que como dentro do **if** temos somente um comando `soma = soma + num`, não é necessário colocá-lo entre chaves.

5.4 Exercícios Recomendados

A solução para esses e outros exercícios você encontra na lista de exercícios em `http://www.ime.usp.br/~macmulti/exercicios/inteiros/index.html`.

Procure fazer os exercícios antes de consultar a solução.

1. Dizemos que um número natural é triangular se ele é produto de três números naturais consecutivos.
   Exemplo: 120 é triangular, pois 4.5.6 = 120.
   Dado um inteiro não-negativo n, verificar se n é triangular.

6 Sobre a divisão e resto de números inteiros

Ronaldo F. Hashimoto e Carlos H. Morimoto

Nesta aula vamos mostrar alguns aspectos dos operadores `/` (divisão inteira) e `%` (resto de divisão inteira) para solução de problemas em Introdução à Computação.

Ao final dessa aula você deverá saber:

- Calcular o resultado de expressões aritméticas que contenham os operadores de divisão e resto com números inteiros.
- Utilizar os operadores de divisão e resto de números inteiros em seus programas.

6.1 Divisão Inteira

Vimos na aula de comandos básicos (aula 3 - Fundamentos), que o resultado de uma divisão inteira é sempre um número inteiro. Assim, o resultado de `3/4` é **zero** (e não `0.75`, que é um número real) e o resultado de `9/2` é `4`, `9/3` é `3`, `9/4` é `2`, e assim por diante. A parte fracionária é simplesmente eliminada (ou truncada ao invés de ser aproximada para um valor inteiro mais próximo).

Agora, o comentário desta aula é para o uso desta operação na resolução de alguns tipos de exercícios em Introdução à Computação.

Existem exercícios em que há a necessidade de manipular os dígitos de um número inteiro. Para este tipo de exercício, é possível arrancar o último dígito de um número inteiro fazendo uma divisão inteira por 10. Por exemplo, `123/10=12` (`123` sem o último dígito). Assim, o comando `a = n / 10` faz com que a variável `a` guarde o conteúdo da variável `n` sem o último dígito.

6.2 Resto de uma Divisão Inteira

Como já vimos em aulas anteriores, a operação `a % b` é o resto da divisão de `a` por `b`. Exemplos:

- `10 % 4 = 2`.
- `3 % 10 = 3`. Observe que `3` é menor que `10`.
- `3 % 2 = 1`.

Existem exercícios em que há a necessidade de se extrair os dígitos de um número inteiro positivo. Para este tipo de exercício, é possível obter o último dígito de um número inteiro calculando o resto de sua divisão inteira por 10. Por exemplo, `123%10=3` (último dígito de `123`). Assim, o comando `a = n % 10` faz com que a variável `a` guarde o último dígito do número guardado na variável `n`.

Outros exercícios podem exigir a verificação se um número inteiro positivo `n` é par ou ímpar. Neste caso, basta verificar se o resto da divisão de `n` por `2` é `0` (zero) ou `1` (um), respectivamente.

Outros exercícios podem exigir a verificação se um número inteiro `n` é divisível ou não por um outro inteiro `b` (diferente de zero). Neste caso, basta verificar se o resto da divisão de `n` por `b` é `0` (zero) ou não, respectivamente.

6.3 Um Exemplo

Como um exemplo, vamos resolver o seguinte exercício:

Dados um número inteiro `n>0`, e um dígito `d` (`0<=d<=9`), determinar quantas vezes `d` ocorre em `n`. Por exemplo, `3` ocorre `2` vezes em `63453`.

A sequência a ser gerada aqui é composta pelos dígitos de `n`. Para cada dígito `b` de `n`, verificar se `b` é igual a `d`. Em caso afirmativo, incrementar um contador.

Para obter um dígito `b` de `n`, vamos calcular o resto da divisão de `n` por 10, ou seja, `b = n%10`.

Para gerar a sequência composta pelos dígitos de `n`, vamos usar então a seguinte repetição:

```
1       while (n>0) {
2         b = n % 10; /* último dígito de n */
3         n = n / 10; /* arranca último dígito de n */
4       }
```

Na linha 2, a variável `b` recebe o último dígito de `n`. Na linha 3, atualizamos a variável `n` para guardar seu conteúdo antigo sem o último dígito, para que na próxima vez, a variável `b` fique com um outro dígito de `n`. A condição do `while` garante que quando `n` for igual a zero, a sequência de todos os dígitos de `n` já foi gerada.

Assim, para terminar, basta colocar um contador e um `if` para verificar se `b` é igual a `d`:

```
1       printf ("Entre com n>0: ");
2       scanf ("%d", &n);
3       printf ("Entre com d (0<=d<=9): ");
4       scanf ("%d", &d);
5       count = 0;
6       while (n>0) {
7         b = n % 10; /* último dígito de n */
8         if (b == d) {
9           count = count + 1;
10        }
11        n = n / 10; /* arranca último dígito de n */
12      }
13      printf ("%d ocorre %d vezes em %d\n", d, count, n);
```

Tem um erro aí em cima! Você saberia detectar? Sempre quando o fluxo do programa sai do laço, o valor da variável `n` é igual a zero! Pois, caso contrário, o fluxo do programa ainda estaria dentro do laço. Neste caso, o último `printf` sempre vai imprimir o valor zero para a variável `n`. Como corrigir isso?

6.4 Exercícios que Usam estes Conceitos

1. Dado um inteiro `n>0`, calcular a soma dos dígitos de `n`.
2. Dado um número natural na base binária, transformá-lo para a base decimal.
   Exemplo: Dado `10010` a saída será `18`, pois
   $$1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 18.$$
3. Dado um número natural na base decimal, transformá-lo para a base binária.
   Exemplo: Dado `18` a saída deverá ser `10010`.
4. Qualquer número natural de quatro algarismos pode ser dividido em duas dezenas formadas pelos seus dois primeiros e dois últimos dígitos.
   Exemplos:
   - `1297`: `12` e `97`.

- `5314`: `53` e `14`.

Escreva um programa que imprime todos os milhares (4 algarismos) cuja raiz quadrada seja a soma das dezenas formadas pela divisão acima.

Exemplo: raiz de `9801` = `99` = `98` + `01`.

Portanto `9801` é um dos números a ser impresso.

5. Dizemos que um número natural `n` é palíndromo se

   (a) o primeiro algarismo de `n` é igual ao seu último algarismo,

   (b) o segundo algarismo de `n` é igual ao penúltimo algarismo,

   (c) e assim sucessivamente.

   Exemplos:

   - `567765` e `32423` são palíndromos.
   - `567675` não é palíndromo.

   Dado um número natural `n>10` , verificar se `n` é palíndrome.

6. Dados `n>0` e uma sequência de `n` números inteiros, determinar a soma dos números pares.

7. Dados `n>0` e dois números inteiros positivos `i` e `j` diferentes de `0`, imprimir em ordem crescente os `n` primeiros naturais que são múltiplos de `i` ou de `j` e ou de ambos.

8. Dados dois números inteiros positivos, determinar o máximo divisor comum entre eles usando o algoritmo de Euclides.

   Exemplo:

	1	1	1	2
24	15	9	6	3
9	6	3	0	

$= \text{mdc}(24,15)$

9. Dizemos que um número `i` é congruente módulo `m` a `j` se `i % m = j % m`.

   Exemplo: `35` é congruente módulo `4` a `39`, pois `35 % 4 = 3 = 39 % 4`.

   Dados inteiros positivos `n`, `j` e `m`, imprimir os `n` primeiros naturais congruentes a `j` módulo `m`.

7 Mais Detalhes da Linguagem C

Ronaldo F. Hashimoto e Carlos H. Morimoto

Nesta aula vamos mostrar alguns conceitos e detalhes da linguagem C, como blocos, abreviaturas, atribuições e constantes. Ao final dessa aula você deverá ser capaz de:

- Identificar blocos, situações que necessitam a utilização de chaves em seu programa, e situações onde as chaves podem ser omitidas.
- Simular e utilizar abreviaturas do comando de atribuição (como += e -=).
- Simular e utilizar atribuições múltiplas e no instante da declaração.
- Definir constantes e utilizá-las em seus programas.

Observação: por permitirem um programa mais conciso, alguns programadores acreditam que seja mais fácil ler e entender programas que se utilizam desses detalhes. Você não precisa utilizá-los, mas deve entendê-los. Se quiser utilizá-los, use de seu bom senso e utilize esses detalhes em seu programa sempre que você acreditar que vale a pena. Procure não utilizá-los de forma aleatória, e tente sempre ser consistente e claro na organização de seus programas.

7.1 Definição de Bloco

Os comandos dentro de uma repetição (comando `while`) são colocados entre chaves, como mostra a figura 9a.

Comando Composto (bloco)	Comando Simples

while (<condição>) {
  <comando_1>;
  <comando_2>;
  ...
  <comando_n>;
}
 	
<comando_fora_do_while_1>;
	while (<condição>) {
	  <comando_1>;
	
	<comando_fora_do_while_1>;
	...
	<comando_fora_do_while_n>;

(a)	(b)

Figura 9: a) Comando composto definido entre { e }; (b) Comando simples, não é necessário o uso de chaves.

Todos os comandos que estiverem entre as chaves { e } estão dentro da repetição e formam um comando composto.

Agora, se dentro da repetição houver somente um comando, é possível omitir as chaves. Assim, no trecho de programa da figura 9b somente o <comando_1> está dentro da repetição. Os demais comandos estão fora! Note a correta endentação dos comandos.

O mesmo acontece com os comandos `if` e `if-else`. Compare os dois trechos de códigos:

```
num = 1;
soma = 0;
while (num<n) {
  if (n % num == 0) {
    soma = soma + num;
  }
  num = num + 1;
}
if (soma == n) {
  printf ("numero perfeito\n");
}
else {
  printf ("nao eh perfeito\n");
}
```

```
num = 1;
soma = 0;
while (num<n) {
  if (n % num == 0)
    soma = soma + num;
  num = num + 1;
}
if (soma == n)
  printf ("numero perfeito\n");
else
  printf ("nao eh perfeito\n");
```

Estes dois trechos são equivalentes, ou seja, produzem o mesmo resultado. No entanto, no trecho do lado esquerdo, todas chaves foram colocadas; enquanto que no trecho direito, algumas chaves foram omitidas para os comandos simples. Confuso? Basta se lembrar que um comando composto pode ter zero ou mais comandos, ou seja, um comando simples entre chaves nada mais é que um comando composto com apenas um comando.

Note que, uma vez que dentro do comando de repetição (`while`) temos dois comandos:

- um de seleção (`if (n % num == 0)`)
- e outro de atribuição (`num = num + 1`)

o uso das chaves é obrigatório.

EXERCÍCIO: Simule a execução desse programa para n = 120. O que é um número perfeito?

DICA: É muito importante adotar um bom espaçamento entre blocos para que você consiga visualizar claramente a estrutura de seu programa, ou leiaute. Para saber mais dicas sobre leiaute, veja as notas de aula do Prof. Feofiloff em `http://www.ime.usp.br/~pf/algoritmos/aulas/layout.html`.

7.2 Abreviaturas do Comando de Atribuição

Na linguagem C, é possível abreviar alguns comandos de atribuição.

7.2.1 Incremento e Decremento

É normal encontramos comandos de repetição onde é necessário o incremento ou decremento de uma variável, em geral, relacionado à condição de parada da repetição. No exemplo anterior, tivemos a atribuição

```
num = num + 1;
```

que significa aumentar de um o conteúdo da variável `num`. Na linguagem C é possível abreviar este comando fazendo:

```
num++;
```

Exemplos de outras abreviaturas:

Descrição	Exemplo	Comando Abreviado
Incremento de um	`i = i + 1`	`i++`
Decremento de um	`n = n - 1`	`n--`
Incremento por uma variável	`soma = soma + num`	`soma += num`
Decremento por uma vairável	`soma = soma - num`	`soma -= num`
	`mult = mult * num`	`mult *= num`
	`divd = divd / num`	`divd /= num`
	`rest = rest % num`	`rest %= num`

Usando estas abreviaturas, o trecho de programa anterior poderia ser escrito como:

```
num = 1;
soma = 0;
while (num<n) {
  if (n % num == 0)
    soma = soma + num;
  num = num + 1;
}
if (soma == n)
  printf ("numero perfeito\n");
else
  printf ("nao eh perfeito\n");
```

```
num = 1;
soma = 0;
while (num<n) {
  if (n % num == 0)
    soma += num;
  num++;
}
if (soma == n)
  printf ("numero perfeito\n");
else
  printf ("nao eh perfeito\n");
```

7.2.2 Atribuição Múltipla

Em certas situações, podemos ter atribuições de um mesmo valor para várias variáveis. No exemplo abaixo, o valor 0 (zero) pode ser atribuido às variáveis `i`, `soma` e `count` usando as seguintes formas:

```
i = 0;
soma = 0;
count = 0;
```

```
i = soma = count = 0;
```

Para entender o que ocorre no caso de atribuições múltiplas mostradas na coluda da direita, basta observar que o operador de atribuição `=` "devolve" o valor atribuido a uma variável, e sua precedência é da direita para esquerda. Assim, a atribuição `count = 0;` "devolve" o valor zero para ser atribuido a `soma`, que por sua vez "devolve" o mesmo valor para ser atribuido para `i`.

7.2.3 Atribuição na Declaração de Variáveis

Em alguns programas é necessário inicializar algumas variáveis. Por exemplo, no trecho de programa anterior, inicializar as variáveis `soma` e `num`.

É possível fazer estas inicializações destas variáveis na declaração das mesmas. Por exemplo:

```c
# include <stdio.h>

int main () {
  int num, soma, n;

  printf ("Entre com n > 0: ");
  scanf ("%d", &n);

  num = 1;
  soma = 0;
  while (num<n) {
    if (n % num == 0)
      soma += num;
    num++;
  }
  if (soma == n)
    printf ("numero perfeito\n");
  else
    printf ("nao eh perfeito\n");

  return 0;
}
```

```c
# include <stdio.h>

int main () {
  int num = 1, soma = 0, n;

  printf ("Entre com n > 0: ");
  scanf ("%d", &n);

  while (num<n) {
    if (n % num == 0)
      soma += num;
    num++;
  }
  if (soma == n)
    printf ("numero perfeito\n");
  else
    printf ("nao eh perfeito\n");

  return 0;
}
```

Note que no lado esquerdo, a inicialização das variáveis ocorre nos comandos de atribuição imediatamente anterior ao comando de repetição; enquanto que no lado direito, a inicialização das variáveis é feita na declaração das mesmas.

7.3 Definição de Constantes

É possível definir constantes na linguagem C. Uma constante em um programa é um valor que não muda durante a sua execução. Diferente de uma variável, que durante a execução do programa pode assumir diferentes valores.

Vamos começar dando um exemplo. Suponha que no seu programa, você queira definir uma constante de nome `UM` que seja o número inteiro um e uma outra constante de nome `ZERO` que seja o número inteiro zero.

Aí, é possível fazer a seguinte atribuição:

```
num  = UM;
soma = ZERO;
```

Para definir uma constante, você deve fazer:

```
# define <NOME_DA_CONSTANTE> <VALOR>
```

Assim, para nosso exemplo, para definir `UM` e `ZERO` devemos fazer

```
# define UM 1
# define ZERO 0
```

Um programa completo com as constantes pode ser visto na figura 10.

```c
# include <stdio.h>

# define UM 1
# define ZERO 0

int main () {
  int num = UM, soma, n;
  printf ("Entre com n > 0: ");
  scanf ("%d", &n);

  soma = ZERO;
  while (num<n) {
    if (n % num == ZERO)
      soma += num;
    num++;
  }
  if (soma == n)
    printf ("numero perfeito\n");
  else
    printf ("nao eh perfeito\n");

  return 0;
}
```

Figura 10: Programa que utiliza constantes.

8 Operadores Lógicos

Ronaldo F. Hashimoto e Carlos H. Morimoto

Nessa aula vamos fazer uma breve revisão sobre expressões aritméticas e relacionais, e introduzir as expressões lógicas. As expressões lógicas serão bastante utilizadas nesse curso para definir condições complexas (por exemplo, que combinam várias expressões relacionais).

Ao final dessa aula você deverá saber:

- Calcular o valor de expressões aritméticas, relacionais e lógicas.
- Utilizar operadores lógicos para a construção de condições complexas em seus programas.

8.1 Expressões Lógicas

Vimos na aula sobre comandos básicos (aula 3 que existem as expressões aritméticas e expressões relacionais. Vamos fazer uma pequena revisão.

8.1.1 Expressões Aritméticas

Expressões aritméticas são expressões matemáticas envolvendo números inteiros, variáveis inteiras, e os operadores `"+"` (soma), `"-"` (subtração), `"/"` (quociente de divisão inteira), `"%"` (resto de uma divisão inteira) e `"*"` (multiplicação).

Nessa primeira parte do curso, o resultado de uma expressão aritmética será sempre um número inteiro. Veremos mais tarde como trabalhar com expressões aritméticas com números reais.

8.1.2 Expressões Relacionais

Expressões relacionais são expressões que envolvem comparações simples envolvendo operadores relacionais `"<"` (menor), `">"` (maior), `"<="` (menor ou igual), `">="` (maior ou igual), `"!="` (diferente), `"=="` (igual). Estas expressões são normalmente usadas como <`condição`> do comando de repetição **while** e do comando de seleção **if** ou **if**−**else**

Uma comparação simples só pode ser feita entre pares de expressões aritméticas da forma:

<`expr_aritmética_01`> <`oper_relacional`> <`expr_aritmética_02`>

onde <`expr_aritmética_01`> e <`expr_aritmética_02`> são expressões aritméticas e <`oper_relacional`> é um operador relacional.

8.1.3 Expressões Lógicas

Agora vamos entrar no tópico desta aula que são as expressões lógicas que são usadas para criar condições mais complexas. Como vimos, uma expressão relacional é uma comparação que só pode ser feita entre pares de expressões aritméticas. É muito comum encontrar situações onde é necessário realizar comparações que envolvam duas expressões relacionais tais como verificar se o conteúdo de uma variável `x` é positivo (ou seja, `x`>0 - primeira expressão relacional) e ao mesmo tempo menor que 10 (ou seja, `x`<10 - segunda expressão relacional). Nesses casos vamos precisar utilizar os operadores lógicos && (operador `"e"`) e || (isto mesmo,

duas barras verticais - operador "ou"). Note que o resultado de uma expressão lógica pode ser **verdadeiro** ou **falso**.

Observação:

Para verificar se o conteúdo de uma variável x é positivo (ou seja, x>0) e ao mesmo tempo menor que 10 (ou seja, x<10), **NÃO** não é correto em C escrever esta condição como $0 < x < 10$. O correto é utilizar o operador lógico && da seguinte forma (x>0) && (x<10).

Para entender porque, lembre-se que o computador "calcula" os valores de expressões, sejam elas aritméticas, relacionais ou lógicas. O valor de uma expressão é calculada segundo a precedência dos operadores. Como os operadores relacionais nesse caso tem mesma precedência, eles seriam calculados da esquerda para direita, ou seja, primeiro o computador testaria (x>0) e o resultado seria então utilizado para calcular (x<10). O problema é que o resultado de (x>0) é verdadeiro ou falso, e não faz sentido dizer se verdadeiro ou falso é menor que 10.

8.1.4 Operador Lógico &&

A tabela verdade para o operador lógico && é dada abaixo:

<expr_aritmética_01>	<expr_aritmética_02>	<expr_aritmética_01> && <expr_aritmética_02>
verdadeiro	verdadeiro	verdadeiro
verdadeiro	falso	falso
falso	verdadeiro	falso
falso	falso	falso

Observando a tabela acima, podemos concluir que o resultado do operador && é verdadeiro **APENAS** quando os dois operandos (<expr_aritmética_01> e <expr_aritmética_02>) tiverem valor verdadeiro. Se algum deles, ou ambos, tiverem valor falso, o resultado de toda expressão lógica é falso.

É verdade que basta um dos operandos (<expr_aritmética_01> ou <expr_aritmética_02>) ser falso para que toda a expressão tenha um resultado falso.

8.1.5 Operador Lógico ||

A tabela verdade para o operador lógico || é dada abaixo:

<expr_aritmética_01>	<expr_aritmética_02>	<expr_aritmética_01> || <expr_aritmética_02>
verdadeiro	verdadeiro	verdadeiro
verdadeiro	falso	verdadeiro
falso	verdadeiro	verdadeiro
falso	falso	falso

Observando a tabela acima, podemos concluir que o resultado do operador || é falso **APENAS** quando os dois operandos (<expr_aritmética_01> e <expr_aritmética_02>) tiverem valor falso. Se algum deles, ou ambos, tiverem valor verdadeiro, o resultado de toda expressão lógica é verdadeiro.

É verdade que basta um dos operandos (<expr_aritmética_01> ou <expr_aritmética_02>) ser verdadeiro para que toda a expressão tenha um resultado verdadeiro.

8.1.6 Exemplos de Expressões Lógicas

Por exemplo, assuma os seguintes valores para as variáveis inteiras x=1 e y=2. Então, veja o resultado das seguintes condições:

Expressão Lógica	Resultado
x >= y	falso
(x < y)	verdadeiro
x >= (y − 2)	verdadeiro
(x > 0) && (x < y)	verdadeiro
(y > 0)&&(x > 0)&&(x > y)	falso
(x < 0) || (y > x)	verdadeiro
(x < 0)||(y > x)&&(y < 0)	falso
para x=1 e y=2.	

8.1.7 Precedências

Os operadores lógicos também têm precedência, como descrito na tabela abaixo:

Operador Aritmético	Associatividade
*, /, %	da esquerda para a direita
+, -	da esquerda para a direita
<, >, <=, >=	da esquerda para a direita
==, !=	da esquerda para a direita
&&	da esquerda para a direita
||	da esquerda para a direita

Agora, observe a seguinte expressão lógica:

$$2 + x < y * 4 \;\&\&\; x - 3 > 6 + y / 2$$

É claro que alguém que soubesse as precedências de todos os operadores envolvidos saberia em que ordem as operações acima devem ser feitas. Agora, a legibilidade é horrível, não? Assim, uma boa dica para melhorar a legibilidade de seus programas é utilizar parênteses para agrupar as expressões aritméticas, mesmo conhecendo a precedência dos operadores, como por exemplo:

$$((2 + x) < (y * 4)) \;\&\&\; ((x - 3) > (6 + y / 2))$$

que fica muito mais fácil de se entender.

8.1.8 Exemplos em Trechos de Programa

Agora vamos ver algumas situações onde é comum utilizar expressões lógicas em programação. Considere o seguinte trecho de programa:

```
1  printf ("Entre com n > 0: ");
2  scanf ("%d", &n);
3  printf ("Entre com i > 0 e j > 0: ");
4  scanf ("%d %d", &i, &j);
5  if (n % i == 0 || n % j == 0) {
6    printf ("%d\n", n);
7  }
```

Este programa lê três números inteiros positivos `n`, `i` e `j` pelo teclado e verifica se o `n` é divisível ou por `i` ou por `j`. Em caso afirmativo, o conteúdo da variável `n` é impresso na tela. “Eta trecho de programa inútel!”. Mas serve para exemplificar um uso do operador lógico ||.

Agora, vamos para um exemplo mais “legal”. Considere o seguinte trecho de programa:

```
i=0; par=0;
while (i<10 && par == 0) {
  printf ("Entre com n > 0: ");
  scanf ("%d", &n);
  if (n % 2 == 0) {
    par = 1;
  }
  i = i + 1;
}
```

Esta repetição lê uma sequência numérica composta de no máximo 10 (dez) números inteiros positivos. Agora, se o usuário entrar somente com números ímpares, então serão lidos os 10 números ímpares e a variável `par` vai terminar com valor 0 (observe que a variável `par` foi inicializada com zero antes da repetição. Agora, se o usuário digitar algum número par, então a repetição quando retornar para verificar a condição `i<10 && par == 0` terminará (por que?) e a variável `par` terminará com valor 1 quando sair do laço.

8.1.9 Exercício

Dados `n>0` e dois números inteiros positivos `i` e `j` diferentes de 0, imprimir em ordem crescente os `n` primeiros naturais que são múltiplos de `i` ou de `j` e ou de ambos.

Exemplo: Para `n=6`, `i=2` e `j=3` a saída deverá ser : 0, 2, 3, 4, 6, 8.

Estude o trecho de código abaixo e veja se ele é uma solução para o problema acima:

```
printf ("Entre com n>0: ");
scanf ("%d", &n);
printf ("Entre com i>0: ");
scanf ("%d", &i);
printf ("Entre com j>0: ");
scanf ("%d", &j);
cont = 0; cont_num = 0;
while (cont_num < n) {
  if (cont % i == 0 || cont % j == 0) {
    printf ("%d\n", cont);
    cont_num = cont_num + 1;
  }
  cont = cont + 1;
}
```

9 Dicas de Programação

Ronaldo F. Hashimoto e Leliane N. de Barros

Este texto contém algumas dicas de programação para resolução de exercícios do curso de Introdução à Programação.

Ao final dessa aula você deverá saber:

- Definir o conceito de padrão em computação
- Utilizar os padrões computacionais: **padrão sequência númerica lida**, **padrão sequência numérica gerada**, **padrão sequência numérica selecionada** e **padrão sequência numérica alternadamente selecionada**.

9.1 Sequências

A solução da maioria dos exercícios de Introdução à Programação envolve gerar e/ou ler pelo teclado uma sequência numérica. Exemplos:

1. Uma sequência de números inteiros diferentes de zero, terminada por um zero: `2, -3, 7, 1, 2, 0`.
2. A sequência dos números inteiros de `1` a `n`: `1,2,...,n`.
3. Uma sequência com `n>0` números inteiros: para `n=5`, a sequência `-2, 3, 0, -2, 7`.

Note que uma sequência numérica pode ser **gerada** pelo seu programa ou **lida pelo teclado** (digitada pelo usuário). Nos Exemplos 1 e 3 acima, as sequências podem ser lidas pelo teclado; enquanto que no Exemplo 2, a sequência pode ser gerada.

Considere agora os seguintes exercícios de Introdução à Computação:

1. **Exercício:** Dada uma sequência de números inteiros diferentes de zero, terminada por um zero, calcular a sua soma.

   Neste exercício a sequência numérica é uma "sequência de números inteiros diferentes de zero que termina com o número zero" (por exemplo: `2, -3, 7, 1, 2, 0`) que é fornecida pelo usuário, ou seja, o seu programa deve ler esta sequência, número a número, pelo teclado. Note que o número zero não faz parte da sequência; ele somente indica o seu término.

   Dizemos neste caso que a sequência é lida pelo teclado.

2. **Exercício:** Dado um número inteiro `n>0`, determinar a soma dos dígitos de `n`. Por exemplo, a soma dos dígitos de 63453 é 21.

   Neste exercício, a sequência numérica é composta pelos dígitos de `n`. Neste caso, o seu programa deve ler pelo teclado o número `n` e a partir dele gerar cada número da sequência (um dígito de `n`) e acumulá-lo em uma soma.

3. **Exercício:** Dado um inteiro `n>0`, calcular a soma dos divisores positivos de `n`.

   Note que neste exercício, a sequência numérica é composta pelos divisores positivos de `n`. Neste caso, o seu programa deve ler `n` pelo teclado, gerar cada número da sequência e acumulá-lo em uma soma.

Em resumo,

> para resolver um problema de Introdução à Computação, você tem que ter a habilidade de identificar que sequência númerica seu programa tem que gerar ou ler pelo teclado.

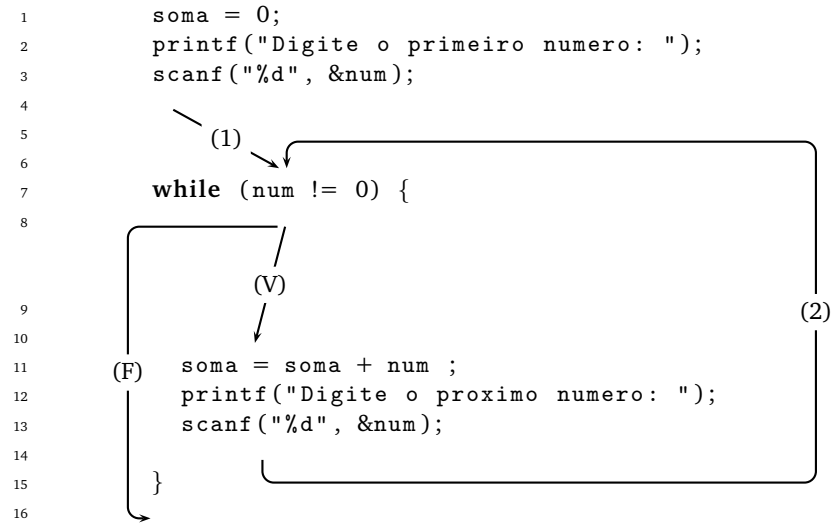
Mais exemplos:

1. **Exercício:** Dado um número inteiro `n>=0`, determinar o seu fatorial.
   Neste exercício, a sequência numérica é composta pelos números inteiros `1,2,3,...,n`.
2. **Exercício:** Dados dois inteiros `x` e `n>0`, calcular a x^n.
   Note que neste exercício, a sequência numérica é a sequência composta por `n` números: $x, x, \ldots, x$.

9.2 Geração e/ou Leitura de uma Sequência Numérica

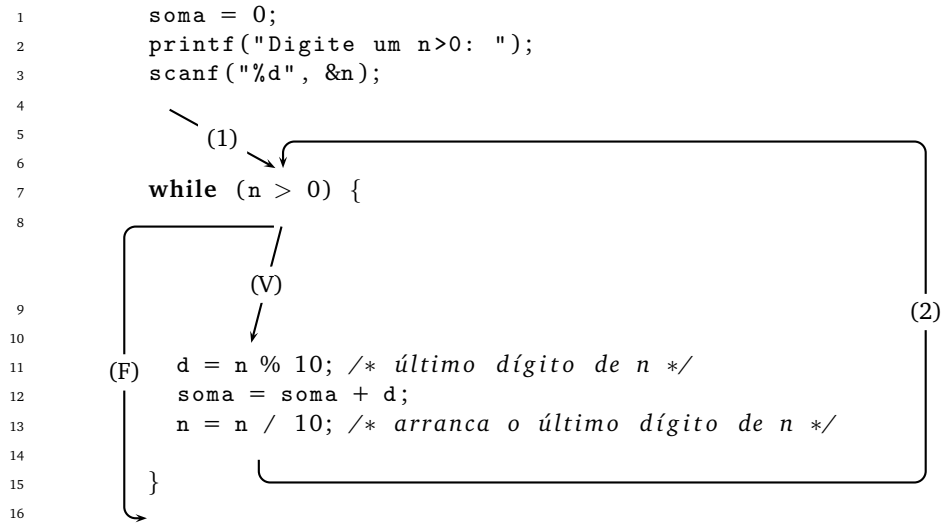
Para gerar e/ou ler uma sequência numérica o seu programa deve usar um comando de repetição. Exemplos:

1. **Exercício:** Dada uma sequência de números inteiros diferentes de zero, terminada por um zero, calcular a sua soma. Neste caso, a sequência deve ser **lida do teclado** usando o comando `scanf` (na linha 13) dentro de uma repetição `while` (linha 7):



   É utilizado o comando de repetição `while` para ler uma sequência de números inteiros pelo teclado e a cada número lido, este número é acumulado em uma variável soma.

2. **Exercício:** Dado um inteiro `n>0`, calcular a soma dos dígitos de `n`. Neste caso, a sequência deve ser **gerada** pelo programa (na linha 11) dentro de uma repetição `while` (linha 7):



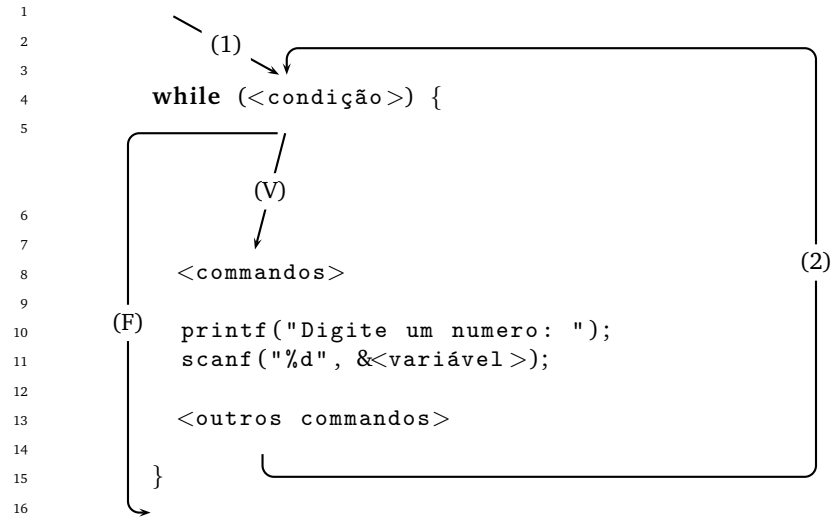
É utilizado o comando de repetição `while` para gerar a sequência dos dígitos de `n` e a cada dígito "descascado", este número é acumulado em uma variável soma.

Em resumo,

> para gerar e/ou ler pelo teclado uma sequência númerica, seu programa deve utilizar um comando de repetição.

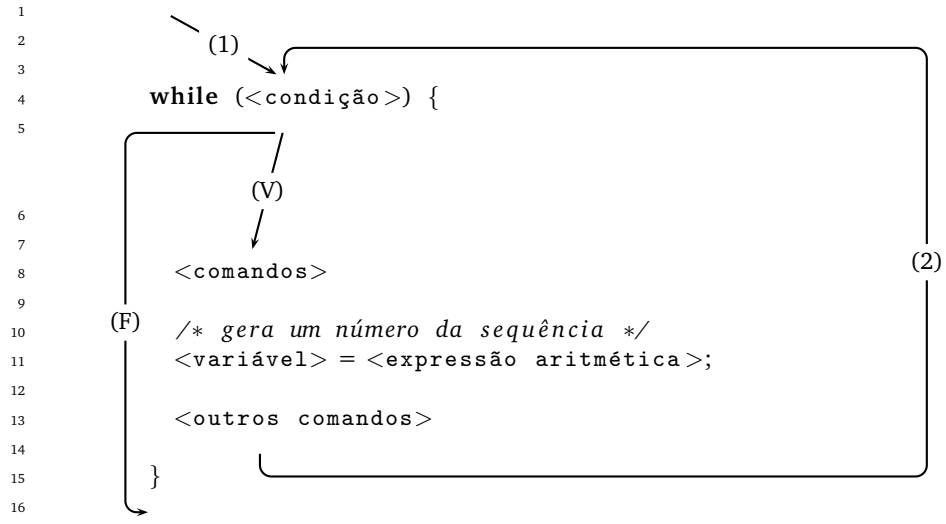
9.3 Padrão de Programação

Um padrão de programação é um trecho de código que tem uma utilidade bem definida e pode ser reutilizado. Por exemplo, o seguinte trecho de código é um padrão para ler uma sequência númerica pelo teclado:



Toda vez que você precisar ler uma sequência numérica pelo teclado, você vai utilizar um trecho de código como o apresentado anteriormente: uma repetição (`while` na linha 4) e um comando de leitura (`scanf` na linha 11) dentro desta repetição. Vamos chamar este padrão de **padrão sequência númerica lida**.

O padrão de programação para gerar uma sequência numérica é dado a seguir:



Toda vez que você precisar gerar uma sequência numérica, você vai utilizar um trecho de código como o apresentado anteriormente: uma repetição (`while` na linha 4) e um comando de atribuição que armazena em uma variável o resultado de uma expressão aritmética (`scanf` na linha 11) dentro desta repetição. Vamos chamar este padrão de **padrão sequência numérica gerada**.

Em resumo,

> apresentamos dois padrões de programação, um para ler uma sequência numérica pelo teclado e outro para gerar uma sequência numérica.

9.3.1 Exercícios que Usam Estes Padrões

1. Dado um número inteiro positivo `n`, calcular a soma dos `n` primeiros números naturais.
2. Dados um inteiro `x` e um inteiro não-negativo `n`, calcular x^n.
3. Dado um inteiro não-negativo `n`, determinar `n!`
4. Um matemático italiano da idade média conseguiu modelar o ritmo de crescimento da população de coelhos através de uma sequência de números naturais que passou a ser conhecida como sequência de Fibonacci. O `n`-ésimo número da sequência de Fibonacci F_n é dado pela seguinte fórmula de recorrência:

$$\begin{cases} F_1 = 1 \\ F_2 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \text{para } n > 2 \end{cases}$$

   Faça um programa que, dado `n>0`, calcula F_n.

5. Qualquer número natural de quatro algarismos pode ser dividido em duas dezenas formadas pelos seus dois primeiros e dois últimos dígitos.

   Exemplos:

   - `1297`: `12` e `97`.
   - `5314`: `53` e `14`.

   Escreva um programa que imprime todos os milhares (4 algarismos) cuja raiz quadrada seja a soma das dezenas formadas pela divisão acima.

   Exemplo: raiz de `9801` = `99` = `98` + `01`.

   Portanto `9801` é um dos números a ser impresso.

9.4 Uso dos Comandos de Seleção

Por enquanto, falamos somente do uso do comandos de repetição. E os comandos de seleção (`if` ou `if-else`), como podem ser usados? Para ilustrar a forma de como estes comandos podem ser utilizados, considere o seguinte problema:

Exercício: Dizemos que um inteiro positivo `n` é perfeito se for igual à soma de seus divisores positivos diferentes de `n`. Exemplo: 6 é perfeito, pois 1+2+3 = 6. O problema é: dado um inteiro positivo `n`, verificar se `n` é perfeito.

Você é capaz de detectar qual é a sequência numérica que está envolvida neste exercício?

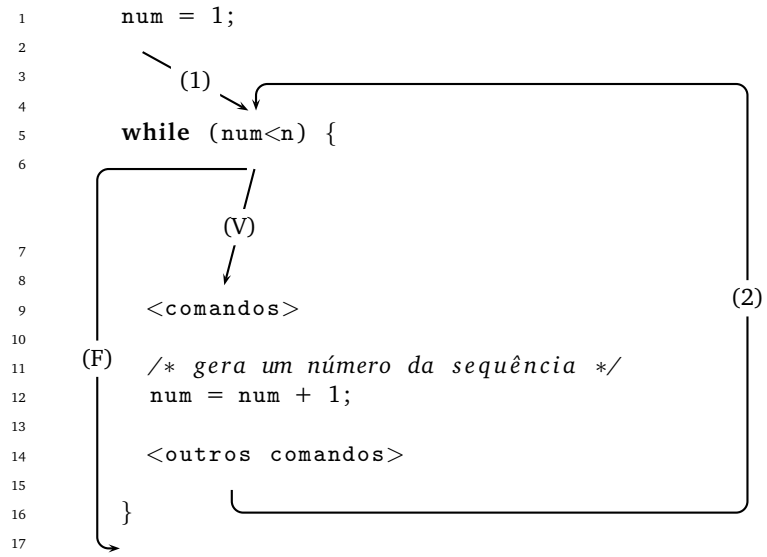
É a sequência dos divisores positivos de `n`. No exemplo acima, a sequência é: `1`, `2` e `3`. Além disso, esta sequência deve ser gerada pelo se programa. Dessa forma, vamos ter que usar o padrão **sequência numérica gerada**.

Agora, vem a pergunta: como gerar a sequência dos divisores positivos de `n`? Você pode notar que não é possível gerar diretamente esta sequência usando o padrão **sequência numérica gerada**!

Uma forma de fazer isso é gerar uma sequência numérica que contenha a sequência dos divisores de `n` e que seja facilmente obtida pelo padrão **sequência numérica gerada**. Esta sequência poderia ser a sequência dos números inteiros de `1` a `n-1`:

$$1, 2, \ldots, n-1$$

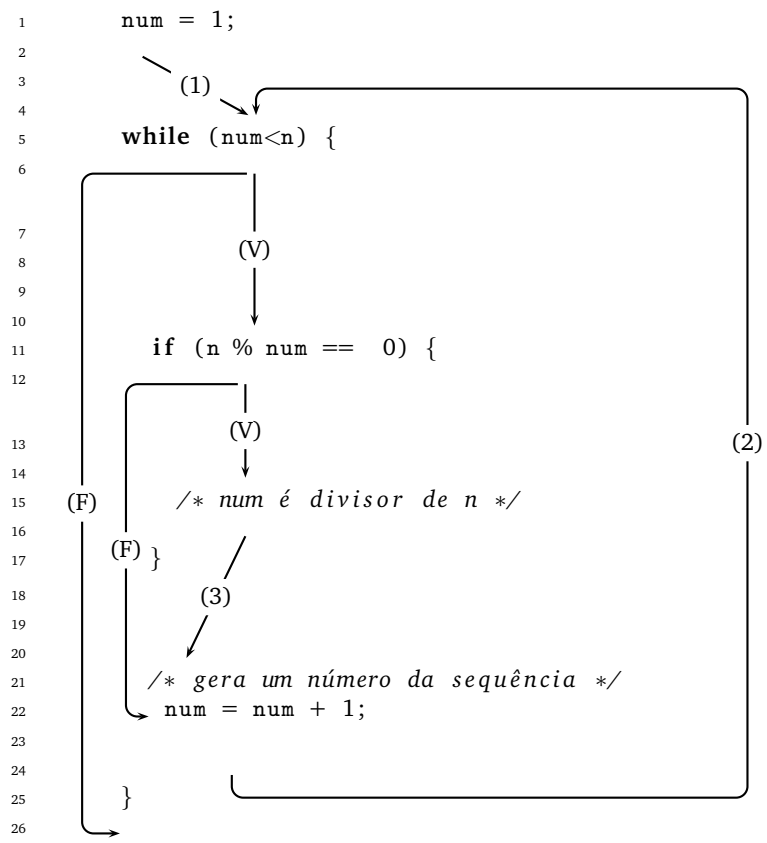
que pode ser facilmente gerada usando o seguinte trecho de código:



Observe bem este trecho de programa. A variável `num` guarda cada número da sequência gerada. Se o conteúdo da variável `num` fosse imprimida dentro do laço, seria gerada a sequência de inteiros de `1` a `n-1`.

Note a condição do `while`. Quando `num` fica igual a `n`, a condição fica falsa e o fluxo do programa sai da repetição (seguindo a seta marcada com `(F)`), garantindo que dentro do laço nunca o conteúdo da variável `num` fica igual ao contéudo da variável `n`. Note também que imediatamente depois que o fluxo do programa sai do laço, o contéudo da variável `num` é sempre igual ao conteúdo da variável `n`, pois caso contrário, o fluxo do programa continuaria dentro do laço.

Depois usando um comando de seleção `if`, "selecionar" os números da sequência gerada que são divisores de `n` da seguinte forma:



Observe então que a repetição (`while` da linha 5) é usada para gerar a sequência de inteiros de `1` a `n-1` e o comando de seleção (`if` da linha 11) é usado para selecionar os números que são divisores de `n`.

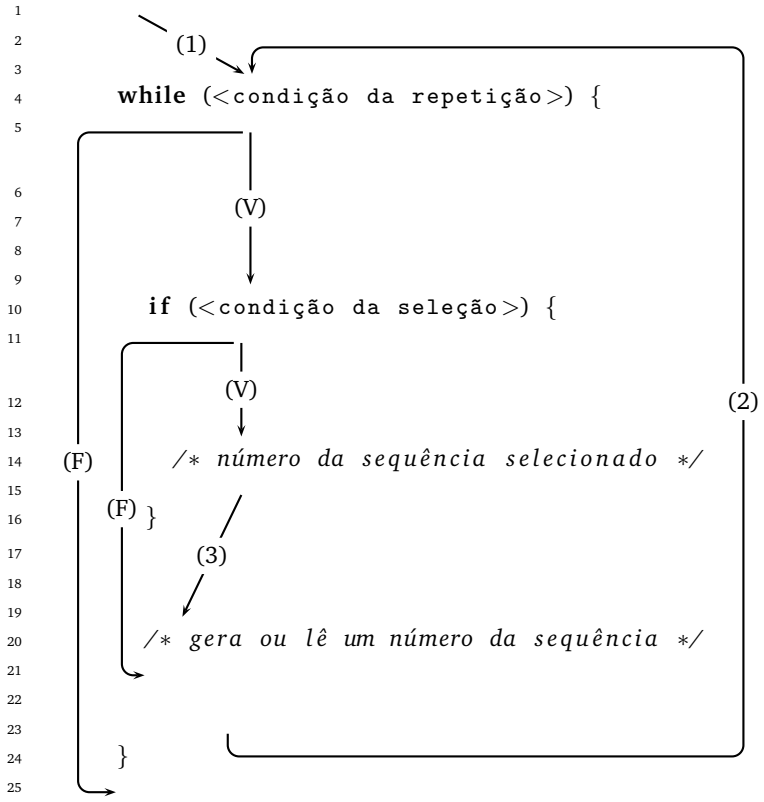
Agora, precisamos acumular em uma variável a soma dos divisores de `n` e depois (fora do laço) testar se a soma final é igual a `n`. Desta forma, teríamos:

```
1   num = 1;
2   soma = 0;
3   while (num<n) {
4     if (n % num ==  0) {
5       /* num é divisor de n */
6       soma = soma + num;
7     }
8     /* gera um número da sequência */
9     num = num + 1;
10  }
11  if (soma == n) {
12    printf ("%d eh um numero perfeito\n", n);
13  }
14  else {
15    printf ("%d nao eh um numero perfeito\n", n);
16  }
```

Fica como tarefa para você escrever o programa completo que seja solução para este exercício!

9.5 Padrão Sequência Numérica Selecionada

Note que podemos então ter um novo padrão de programação que chamamos de **padrão sequência numérica selecionada**:



9.5.1 Exercícios que Usam este Padrão

1. Dado um número inteiro positivo `n`, imprimir os `n` primeiros naturais ímpares. Exemplo: Para n=4 a saída deverá ser 1,3,5,7.

2. Uma loja de discos anota diariamente durante o mês de março a quantidade de discos vendidos. Determinar em que dia desse mês ocorreu a maior venda e qual foi a quantidade de discos vendida nesse dia.

3. Dados `n>0` e uma sequência de `n` números inteiros, determinar a soma dos números pares.

4. Dados `n>0` e dois números inteiros positivos `i` e `j` diferentes de `0`, imprimir em ordem crescente os `n` primeiros naturais que são múltiplos de `i` ou de `j` e ou de ambos.

5. Dados dois números inteiros positivos, determinar o máximo divisor comum entre eles usando o algoritmo de Euclides.

   Exemplo:

	1	1	1	2
24	15	9	6	3
9	6	3	0	

   $= \text{mdc}(24,15)$

6. Dizemos que um número `i` é congruente módulo `m` a `j` se `i % m = j % m`.

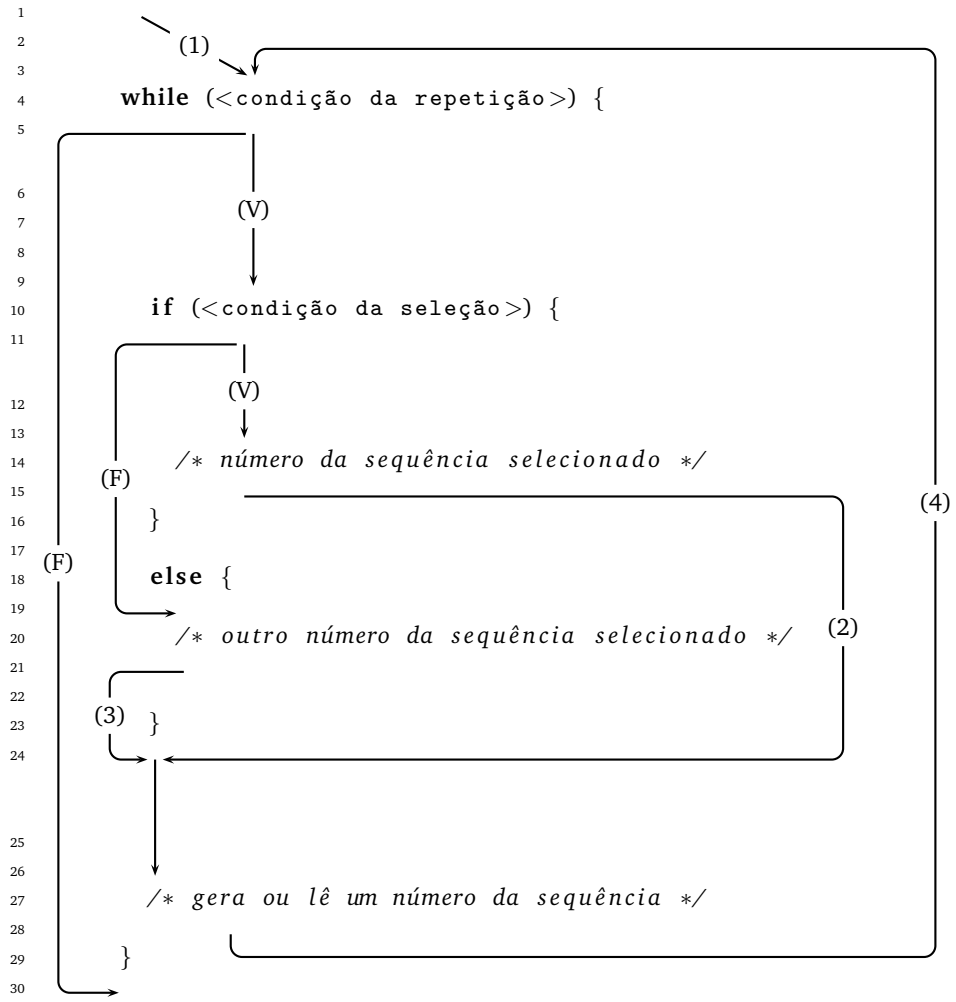
   Exemplo: `35` é congruente módulo `4` a `39`, pois `35 % 4 = 3 = 39 % 4`.

   Dados inteiros positivos `n`, `j` e `m`, imprimir os `n` primeiros naturais congruentes a `j` módulo `m`.

9.6 Padrão Sequência Numérica Alternadamente Selecionada

Note que poderíamos ter situações em que fossem necessários selecionar números de uma sequência alternadamente. Por exemplo, selecionar da sequência de `1` a `n` os números pares e os números ímpares (ou os números são divisores de `3` e os que não são) para detectar quantos são pares e qual é a soma dos ímpares (não sei para que saber estas informações, mas é um bom exemplo de uma sequência alternadamente selecionada).

Este padrão usaria então um comando de seleção `if-else` dentro de uma repetição `while` da seguinte forma:



9.6.1 Exercícios que Usam este Padrão

1. Dados `n>0` e uma sequência de `n` números inteiros positivos, determinar a soma dos números pares e a quantidade dos números ímpares.
2. Dado um inteiro `n>0`, determinar a quantidade de divisores positivos e pares de `n` e calcular a soma dos divisores positivos de `n`.

10 Comando de Repetição for

Ronaldo F. Hashimoto e Carlos H. Morimoto

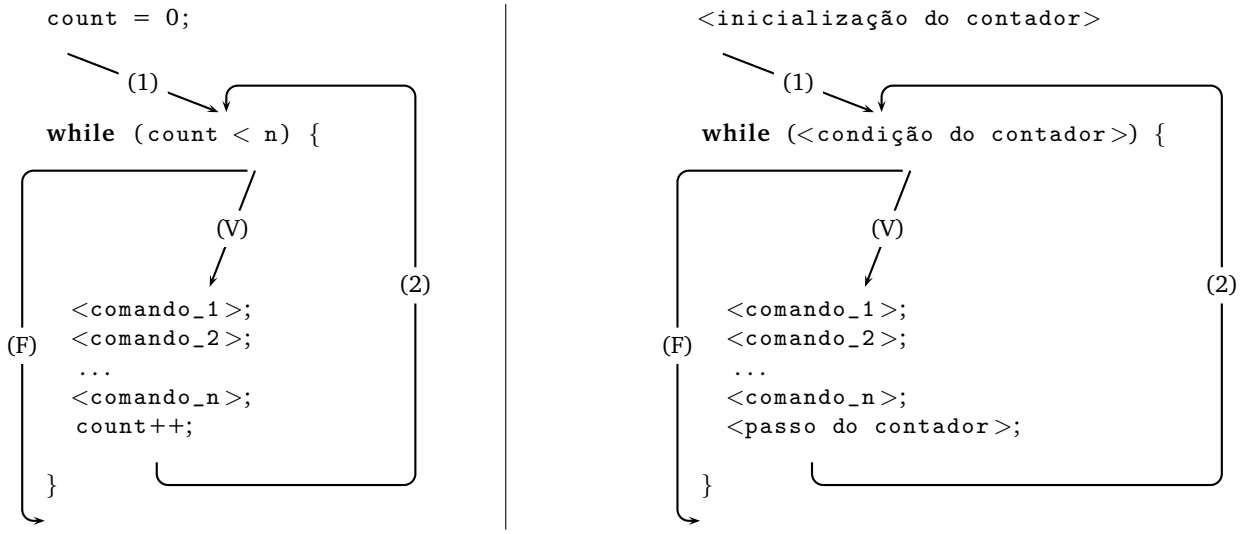
Essa aula introduz o comando de repetição `for`, que pode ser entendida como uma notação compacta do comando `while`.

Ao final dessa aula você deverá saber:

- Descrever a sintaxe do comando `for`.
- Simular o funcionamento do comando `for`.
- Utilizar o comando `for` em programas.

10.1 Motivação

Existem várias situações em que a repetição que resolve um exercício de programação envolve o uso de um contador que começa com um valor inicial (`<inicialização>`) e aumenta/diminui de um em um (ou de dois em dois, a cada passo). Observe atentamente os seguintes trechos de programa:



No trecho do lado esquerdo, a variável `count` controla o número de vezes em que os comandos dentro da repetição são executados. Como `count` é inicializado com zero, aumenta de um em um até chegar em `n`, então os comandos `<comando_1>`, `<comando_2>`, ..., `<comando_n>` são executados `n` vezes.

No trecho do lado direito, nós temos um padrão para este tipo de repetição controlada por um contador. O contador deve ser inicializado antes da repetição; dentro do laço, o contador deve ser incrementado (ou decrementado) de um valor; e a condição do contador deve garantir a parada da repetição. Vamos chamar este padrão de programação de **repetição com contador**. Este padrão de programação pode ser escrito usando o comando `for`.

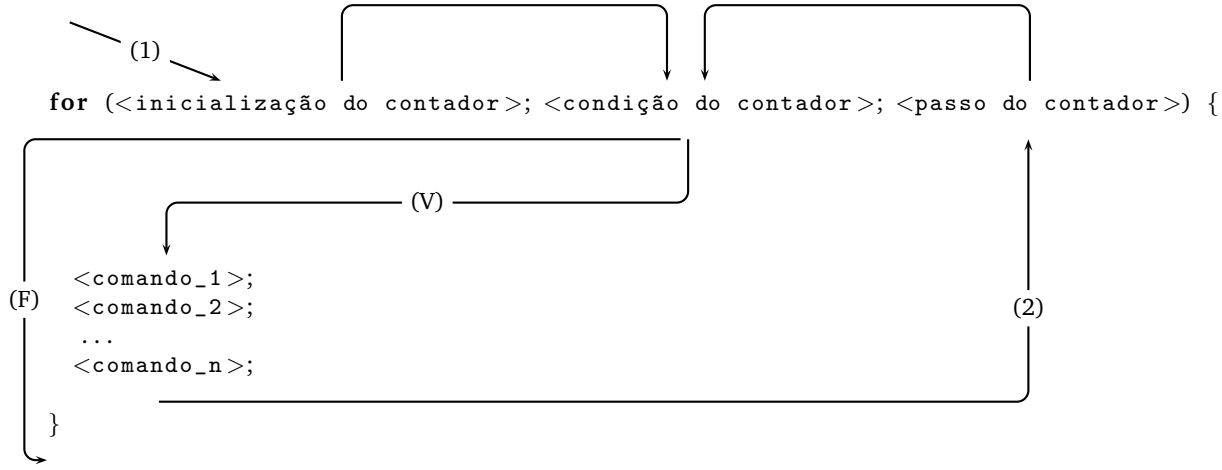
10.2 Sintaxe

O comando `for` é um outro comando de repetição para o padrão de repetição com contador. A sua sintaxe é:

```
for (<inicialização do contador>; <condição do contador>; <passo do contador>) {
  <comando_1>;
  <comando_2>;
  ...
  <comando_n>;
}
```

10.3 Descrição

O fluxo do comando `for` é dado pela seguinte figura:



Basicamente, este comando de repetição tem o seguinte significado: enquanto a <condição> for **verdadeira**, a sequência de comandos <comando_1>, <comando_2>, ..., <comando_n> é executada.

Vamos analisar o "fluxo" do programa usando o comando de repetição **for**. Primeiramente, a execução do programa vem e faz a inicialização do contador (seta marcada com (1)). Depois a <condição do contador> do **for** é testada. Se "de cara" a <condição do contador> é **falsa**, o fluxo do programa ignora a sequência de comandos dentro do **for** e segue a seta marcada com (F). Agora, se a <condição do contador> é **verdadeira**, então o fluxo do programa segue a seta marcada com (V) e executa a sequência de comandos dentro do **for**; executado o último comando (<comando_n>), o fluxo do programa segue a seta marcada com (2) e executa <passo do contador> (aumenta/diminui o contador de passo) e volta a testar a <condição do contador>. Se a <condição do contador> é **verdadeira**, então o fluxo do programa segue a seta marcada com (V) repetindo a sequência de comandos dentro do **for**. Se <condição do contador> é **falsa**, o fluxo do programa ignora a sequência de comandos e segue a seta marcada com (F).

Note que o fluxo de execução do **for** é idêntico ao do **while** do padrão de repetição com contador.

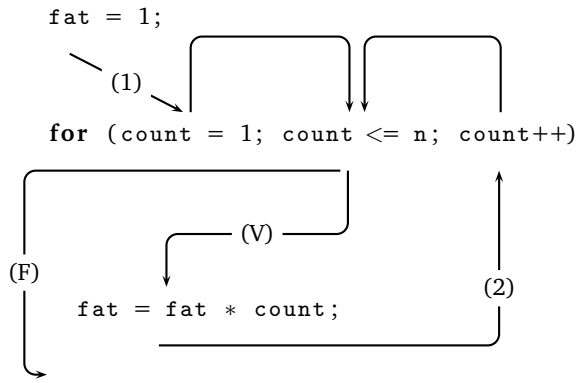
Na verdade, todo comando **for** pode ser escrito usando o comando **while** e vice-versa. A vantagem de usar o comando **for** em repetição com contador é que sua forma é compacta.

10.4 Exemplo

Dado um número inteiro `n>=0`, calcular `n!`

Precisamos gerar a sequência de números $1, 2, \ldots,$ `n`. E acumular a multiplicação para cada número gerado.

Para gerar esta sequência e ao mesmo tempo acumular a multiplicação, vamos usar o comando **for** da seguinte forma:



10.5 Perguntas

1. Este trecho de programa funciona para `n=0`?
2. Por que neste trecho de programa o comando `fat = fat * count;` não está entre chaves?

10.6 Mais um Exercício

Dizemos que um número é triangular se ele é produto de três números naturais consecutivos (e.g.: `120` é triangular pois $120 = 4 \times 5 \times 6$). Dado um natural `n>0`, determinar se `n` é triangular.

Basicamente, precisamos definir 3 coisas:

1. Início: qual o primeiro candidado a solução a ser testado?
2. Fim: a partir de que valor não é mais necessário procurar a solução?
3. Incremento: como gerar a próxima solução a ser testada.

Nesse caso, como os números são positivos, a primeria solução a ser testada seria $1 \times 2 \times 3$. A segunda seria $2 \times 3 \times 4$, e assim por diante. Caso o produto for igual ao número que desejamos testar, então encontramos a solução, e a resposta seria positiva. Quando o produto se torna maior que o número desejado, sabemos que o número não pode ser triangular, e podemos encerrar a busca.

Dessa forma, devemos gerar a sequência $1, 2, 3, \ldots,$ `n` e para cada número `count` desta sequência, calcular o produto `count` $\times$ `(count+1)` $\times$ `(count+2)` e verificar se ele é igual a `n`.

Vejamos um programa que faz isso usando **for**:

```
triangular = FALSE;

for (count=1; count<=n && triangular == FALSE; count++)
  if (count * (count+1) * (count+2) == n)
    triangular = TRUE;

if (triangular == TRUE)
  printf("O numero %d e triangular\n", n);
else
  printf("O numero %d nao eh triangular\n", n);
```

10.7 Perguntas e Comentários

1. `triangular` é um indicador de passagem?
2. O comando **if** dentro do **for** não está entre chaves pois somente tem um comando dentro do **for**.
3. O comando `triangular = TRUE` dentro do **if** não está entre chaves pois somente tem um comando dentro do **if**.
4. Para que serve a condição adicional `triangular == FALSE` dentro do comando **for**?

10.8 Exercícios Recomendados

1. Dado um inteiro `p>1`, verificar se `p` é primo.

   Use indicador de passagem, o comando **for** e interrompa a repetição quando descobrir que `p` não é um número primo.

2. Dados um número inteiro `n>0`, e uma sequência com `n` números inteiros, determinar o maior inteiro da sequência. Por exemplo, para a sequência

   `6, -2, 7, 0, -5, 8, 4`

   o seu programa deve escrever o número `8`.

11 Repetições Encaixadas

Ronaldo F. Hashimoto e Carlos H. Morimoto

O conceito de repetições encaixadas nada mais é que uma repetição dentro de uma outra repetição. Ao final dessa aula você deverá ser capaz de:

- Simular programas com repetições encaixadas.
- Utilizar repetições encaixadas em seus programas.

11.1 Repetições Encaixadas

Até o momento, nossos programas foram simples e os problemas não exigiam muitos comandos dentro de uma repetição. Mas os comandos dentro de uma repetição podem se tornar bastante complexos. Nessa aula, vamos ver como alguns problemas exigem a criação de repetição dentro de repetição (ou repetições encaixadas), como por exemplo:

```
1       while (<condição 1>) {
2         while (<condição 2>) {
3           while (<condição 3>) {
4           }
5         }
6       }
```

A seguir vamos apresentar esse recurso através de um problema.

11.2 Exemplo de um Problema que usa Repetições Encaixadas

Problema: Dados dois naturais `n>0` e `m>0`, determinar entre todos os pares de números inteiros `(x,y)` tais que $0 \leq x \leq n$ e $0 \leq y \leq m$, um par para o qual o valor da expressão $x \times y - x^2 + y$ seja máximo e calcular também este máximo. Em caso de empate, imprima somente um valor.

Exemplo: para `n=2` e `m=3`, a saída do programa deve ser $(1, 3)$ com valor máximo igual a `5`.

Solução:

- Tomando o exemplo `n = 2` e `m = 3`;
- Gerar todos os pontos (x, y) tais que $0 \leq x \leq n$ e $0 \leq y \leq m$.
- Para cada `x` fixo, temos que gerar os valores para `y` de `0` a `m=3`. Assim, temos:

  `x=0`

  $(0,0)$ $(0,1)$ $(0,2)$ $(0,3)$ $\Rightarrow$ Para `x=0`, um laço para gerar os valores de `y` de `0` a `3`.

  `x=1`

  $(1,0)$ $(1,1)$ $(1,2)$ $(1,3)$ $\Rightarrow$ Para `x=1`, um laço para gerar os valores de `y` de `0` a `3`.

  `x=2`

  $(2,0)$ $(2,1)$ $(2,2)$ $(2,3)$ $\Rightarrow$ Para `x=2`, um laço para gerar os valores de `y` de `0` a `3`.
- Assim, para cada `x` fixo, temos uma repetição para gerar os valores para `y` de `0` a `3`.

```
/* para um valor fixo de x */
y = 0;
while (y <= m) {
  printf ("%d, %d\n", x, y);
  y = y + 1;
}
```

- Agora é necessário ter uma repetição para gerar os valores de `x` de `0` a `2`.

```
x = 0;
while (x <= n) {
  /* gera um valor para x */
  x = x + 1;
}
```

- Note que a repetição que gera a sequência para `y` deve estar dentro da repetição que gera a sequência para `x`
- Assim, juntando os códigos, temos

```
x = 0;
while (x <= n) {
  /* gera um valor para x */
  /* para um valor fixo de x */
  y = 0;
  while (y <= m) {
    printf ("%d, %d\n", x, y);
    y = y + 1;
  }
  x = x + 1;
}
```

- Agora temos que detectar um par `(x,y)` para o qual o valor da expressão $\texttt{x} \times \texttt{y} - \texttt{x}^2 + \texttt{y}$ seja máximo e calcular também este máximo.
- Para cada par `(x,y)` gerado pelo código anterior, calcular o valor da expressão $\texttt{x} \times \texttt{y} - \texttt{x}^2 + \texttt{y}$.

`x=0`

$(0,0) \Rightarrow 0 \quad (0,1) \Rightarrow 1 \quad (0,2) \Rightarrow 2 \quad (0,3) \Rightarrow 3$

`x=1`

$(1,0) \Rightarrow -1 \quad (1,1) \Rightarrow 1 \quad (1,2) \Rightarrow 3 \quad (1,3) \Rightarrow 5$

`x=2`

$(2,0) \Rightarrow -4 \quad (2,1) \Rightarrow -1 \quad (2,2) \Rightarrow 2 \quad (2,3) \Rightarrow 5$

- Assim, basta ter uma variável `max` que, para cada par gerado, guarda o valor máximo até o presente momento, de forma que:

`x=0`

$(0,0) \Rightarrow 0, \texttt{max=0} \quad (0,1) \Rightarrow 1, \texttt{max=1} \quad (0,2) \Rightarrow 2, \texttt{max=2} \quad (0,3) \Rightarrow 3, \texttt{max=3}$

`x=1`

$(1,0) \Rightarrow -1, \texttt{max=3} \quad (1,1) \Rightarrow 1, \texttt{max=3} \quad (1,2) \Rightarrow 3, \texttt{max=3} \quad (1,3) \Rightarrow 5, \texttt{max=5}$

`x=2`

$(2,0) \Rightarrow -4, \texttt{max=5} \quad (2,1) \Rightarrow -1, \texttt{max=5} \quad (2,2) \Rightarrow 2, \texttt{max=5} \quad (2,3) \Rightarrow 5, \texttt{max=5}$

- Para cada par gerado, calcula-se o valor da expressão $\texttt{v} = \texttt{x} \times \texttt{y} - \texttt{x}^2 + \texttt{y}$ e testa se $\texttt{v} > \texttt{max}$. Em caso afirmativo, o valor de `max` deve ser atualizado para `v` fazendo $\texttt{max} = \texttt{v}$;
- Temos então o seguinte código:

```
x = 0;
while (x <= n) {
  /* gera um valor para x */
  y = 0;
  while (y <= m) {
    v = x*y - x*x + y;
    if (v > max)
      max = v;
    y = y + 1;
  }
  x = x + 1;
}
printf ("O valor maximo = %d\n", max);
```

- O código anterior consegue determinar o valor máximo, mas não consegue detectar qual par `(x,y)` tem este valor.
- Para detectar qual par `(x,y)` tem o valor máximo, temos que guardá-lo toda vez que a variável `max` é atualizada:

```
x = 0;
while (x <= n) {
  /* gera um valor para x */
  y = 0;
  while (y <= m) {
    v = x*y - x*x + y;
    if (v > max) {
      max = v;
      x_max = x;
      y_max = y;
    }
    y = y + 1;
  }
  x = x + 1;
}
printf ("O valor maximo = %d em x = %d e y = %d\n", max, x_max, y_max);
```

- Falta ainda resolver um problema: quais devem ser os valores iniciais para `max`, `x_max` e `y_max`?
- O problema de iniciarmos `max` com 0 é que se os valores calculados de `v` forem todos negativos, a variável `max` nunca será atualizada.
- O problema de iniciarmos `max` com um número negativo (digamos -1000) é que se os valores calculados de `v` forem todos menores que -1000, a variável `max` nunca será atualizada.
- Uma boa idéia é inicializarmos `max` com um valor que `v` assume. Poderia ser qualquer valor da sequência $0, 1, 2, 3, -1, 1, 3, 5, -4, -1, 2, 5$.
- Assim, podemos escolher um par `(x,y)` que é gerado. Por exemplo, o par $(0, 0)$ e calcular o valor da expressão $v = x \times y - x^2 + y$ para este par. Neste caso, $v = 0$ e colocamos para (x_max, y_max) o par $(0, 0)$.
- A solução final seria:

```c
# include <stdio.h>

int main () {
  int x, y, n, m, v, x_max, y_max;

  printf ("Entre com n>0: ");
  scanf ("%d", &n);

  printf ("Entre com m>0: ");
  scanf ("%d", &m);

  x = 0; max = x_max = y_max = 0;
  while (x <= n) {
    /* gera um valor para x */
    y = 0;
    while (y <= m) {
      v = x*y - x*x + y;
      if (v > max) {
        max = v;
        x_max = x;
        y_max = y;
      }
      y = y + 1;
    }
    x = x + 1;
  }
  printf ("O valor maximo = %d em x = %d e y = %d\n", max, x_max, y_max);
  return 0;
}
```

11.3 Exercícios Recomendados

1. Dados `n>0` números inteiros positivos, calcular a soma dos que são primos.

   (a) Sua solução deve conter uma repetição com contador para ler `n` números pelo teclado.

   (b) Para cada número lido, seu programa deve testar se ele é primo (usando uma outra repetição com indicador de passagem). Em caso afirmativo, acumular em uma soma.

12 Indicador de Passagem

Ronaldo F. Hashimoto, Carlos H. Morimoto e Leliane N. de Barros

`Indicador de Passagem` é um padrão bastante utilizado em computação para identificar a ocorrência de um evento que ajuda no controle do seu programa. O indicador de passagem é uma variável que inicia com um determinado valor, e caso o evento que ele marca ocorra, seu valor é alterado e não muda mais.

Ao final dessa aula você deverá saber:

- Descrever o funcionamento de indicadores de passagem.
- Identificar situações onde indicadores de passagem podem ser utilizados.
- Utilizar indicadores de passagem em seus programas.

12.1 Conceito de Indicador de Passagem

Considere o padrão de programação dado na Fig. 11.

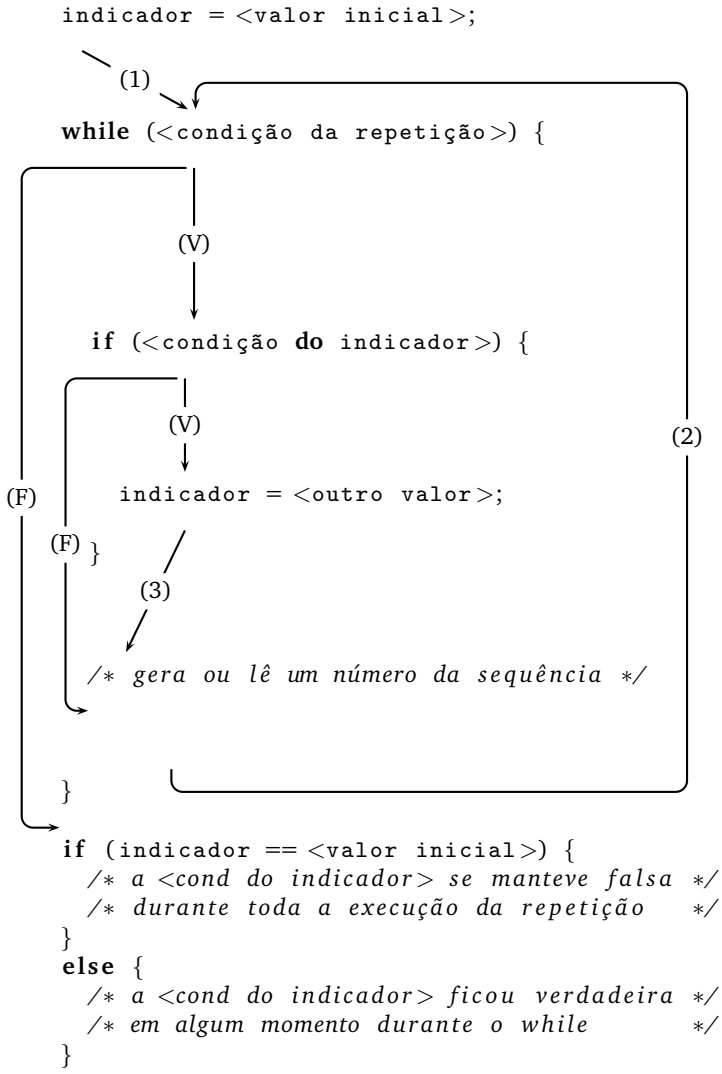


Figura 11: Padrão Indicador de Passagem

Na linha 5, temos uma repetição (`while`) que trata de gerar ou ler pelo teclado uma sequência de números. Observe mais uma vez que os exercícios que estamos lidando sempre há uma sequência de números. Antes da repetição, na linha 1, existe uma inicialização de uma variável `indicador` com um certo valor inicial. Dentro da repetição, na linha 11, existe um comando de seleção simples (`if`) que testa uma propriedade da sequência de números (por exemplo, sequência crescente, sequência com todos números positivos, sequência com todos números pares, etc...). Se a condição `<condição do indicador>` ficar verdadeira em algum momento durante a execução da repetição, então o valor da variável `indicador` recebe outro valor diferente do valor inicial. No final da repetição, testa-se o conteúdo da variável `indicador`. Se conteúdo desta variável é o `<valor inicial>`, então a condição `<condição do indicador>` nunca foi satisfeita durante a execução da repetição. Agora, se o conteúdo é `<outro valor>`, então, em algum momento, durante a execução da repetição, a condição `<condição do indicador>` foi satisfeita.

Vamos chamar este padrão de programação de **padrão indicador de passagem**.

12.2 Exemplo

Considere o seguinte programa que lê uma sequência de dez números inteiros:

```
1   # include <stdio.h>
2
3   int main () {
4     int pos, i, x;
5
6     pos = 0;
7     i = 0;
8     while (i<10) {
9       printf ("Entre com x: ");
10      scanf ("%d", &x);
11      if (x > 0) {
12        pos = 1;
13      }
14      i = i + 1;
15    }
16    if (pos == 0)
17      printf ("Todos elems menores ou iguais a zero\n");
18    else
19      printf ("Pelo menos um elem. maior do que zero\n");
20
21    return 0;
22  }
```

A variável `pos` é um **indicador de passagem**.

Na linha 8, temos uma repetição que trata de ler pelo teclado uma sequência de dez números inteiros. Antes da repetição, na linha 6, existe uma inicialização da variável `pos` com valor inicial igual a um. Dentro da repetição, na linha 11, existe um comando de seleção simples (`if`) que testa se o número lido é maior do que zero. Se esta condição ficar verdadeira em algum momento durante a execução da repetição, então o valor da variável `pos` recebe outro valor diferente do valor inicial, que no caso é o valor um. No final da repetição, testa-se o conteúdo da variável `pos`. Se conteúdo da variável `pos` é o zero, então a condição `x>0` nunca foi satisfeita durante a execução da repetição, indicando que todos os elementos da sequência são menores ou iguais a zero. Agora, se o conteúdo é um, então, em algum momento durante a execução da repetição, a condição `x>0` foi satisfeita, indicando que pelo menos um elemento da sequência é maior do que zero.

É importante notar que o indicador de passagem tenta capturar uma propriedade da sequência. No exemplo anterior, o indicador de passagem tenta capturar se a sequência contém algum número positivo.

Além disso, note que a propriedade da sequência que o indicador de passagem tenta capturar sempre coloca uma questão cuja resposta é “sim” (verdadeira) ou “não” (falsa). No exemplo anterior, o indicador tenta responder a questão: a sequência tem algum número positivo?

12.3 Uso de Constantes

Na aula “Detalhes da Linguagem C” comentamos que é possível definir constantes. Nesta aula, vamos falar um pouco da sua utilidade.

Para responder a questão que o indicador de passagem tenta responder, poderíamos definir duas constantes: `TRUE` e `FALSE`. O indicador de passagem terminaria com valor `TRUE` se a reposta da questão for “sim” e com valor `FALSE` caso contrário.

Assim, o programa anterior ficaria:

```c
# include <stdio.h>

# define TRUE 1
# define FALSE 0

int main () {
  int pos, i, x;

  pos = FALSE;
  i = 0;
  while (i<10) {
    printf ("Entre com x: ");
    scanf ("%d", &x);
    if (x > 0) {
      pos = TRUE;
    }
    i = i + 1;
  }
  if (pos == FALSE)
    printf ("Todos elems menores ou iguais a zero\n");
  else
    printf ("Pelo menos um elem. maior do que zero\n");

  return 0;
}
```

12.4 Exemplo

Considere o seguinte problema:

Dado `n>1`, verificar se `n` tem dois dígitos adjacentes iguais. Exemplos: (1) `n = 21212 => Não` e (2) `n = 212212 => Sim`.

Observe:

- `n = 212123`.
- Sequência Numérica deste exercício: os dígitos de `n`.
- Neste exemplo, `2, 1, 2, 1, 2, 3`.
- Isto significa que dado `n>1`, a sequência é gerada.
- Usar a propriedade de divisão e resto por `10`.
- `n/10` $\Rightarrow$ quociente inteiro de `n` por `10`, ou seja, o número `n` sem o último dígito.
- Então `212123/10 = 21212`
- `n%10` $\Rightarrow$ o resto da divisão de `n` por `10`, ou seja, o último dígito de `n`.
- Então `212123%10 = 3`.
- Usando estas propriedades, vamos gerar a sequência dos dígitos de `n` de trás para frente. Mas isto não tem importância, uma vez que a propriedade de adjacência não depende se a sequência está de trás para frente e vice-versa. Veja, logo depois do código, por que a sequência é inversa.
- Descascar o número `n` até que ele vire zero.
- Neste exercício, queremos verificar uma propriedade da sequência gerada: se ela contém dois números adjacentes iguais.

- Para verificar esta propriedade, vamos usar um indicador de passagem de nome `adjacente` que começaria com valor `FALSE`. Se em algum momento, o programa encontrar dois números adjacentes iguais, então esta variável recebe valor `TRUE`.

Usando o padrão indicador de passagem, então temos o seguinte programa:

```
1   # include <stdio.h>
2
3   # define TRUE 1
4   # define FALSE 0
5
6   int main () {
7     int n;
8     int posterior, anterior;
9     int adjacente;
10
11    printf ("Entre com n > 0: ");
12    scanf ("%d", &n);
13
14    adjacente = FALSE;
15    posterior = -1;
16
17    while (n != 0) {
18      anterior  = posterior;
19      posterior = n % 10;
20
21      if (anterior == posterior) {
22        adjacente = TRUE;
23        n = 0;
24      }
25
26      n = n / 10;
27    }
28
29    if (adjacente == FALSE)
30      printf ("NAO\n");
31    else
32      printf ("SIM\n");
33
34    return 0;
35  }
```

Neste esquema de repetição, na linha `19`, a variável `posterior` recebe o último dígito de `n` e, na linha `26`, `n` fica sendo o número `n` sem o último dígito. Assim, para `n = 213`, teríamos a seguinte tabela de valores para `posterior` e `n`:

posterior	n
?	123
3	12
2	1
1	0

Assim, neste exercício, sequência de dígitos de `n` é gerada de trás para frente. No exemplo acima, a sequência para `n = 123` é então $3, 2, 1$.

12.5 Outro Exemplo

Considere o seguinte problema:

Dado um inteiro `n>0`, e uma sequência de `n` inteiros, calcular a sua soma e verificar se a sequência é estritamente crescente.

Neste exercício, além de calcular a soma, queremos verificar uma propriedade da sequência lida: se ela é estritamente crescente ou não. Para verificar esta propriedade, vamos usar um indicador de passagem de nome `crescente` que começaria com valor `TRUE`. Se em algum momento a sequência deixar de ser crescente, então esta variável recebe valor `FALSE`.

Usando o padrão indicador de passagem, então temos o seguinte programa:

```
1   # include <stdio.h>
2
3   # define TRUE 1
4   # define FALSE 0
5
6   int main () {
7     int cont; /* contador dos elementos da sequencia */
8     int n; /* numero de elementos da sequencia */
9     int soma;
10    int num; /* cada elemento da sequencia */
11    int ant; /* elemento anterior ao num */
12    int crescente;
13
14    printf ("Entre com n>0: ");
15    scanf ("%d", &n);
16
17    printf ("Entre com um num. inteiro da seq.: ");
18    scanf ("%d", &num);
19
20    crescente = TRUE;
21    soma = num;
22
23    cont = 2;
24    while (cont <= n) {
25
26      ant = num;
27
28      printf ("Entre com um num. inteiro da seq.: ");
29      scanf ("%d", &num);
30
31      if (ant >= num)
32        crescente = FALSE;
33
34      soma = soma + num;
35
36      cont = cont + 1;
37    }
38
39    printf ("soma = %d\n", soma);
40
41    if (crescente == TRUE)
42      printf ("Sequencia Estritamente Crescente\n");
43    else
44      printf ("Sequencia Nao Estritamente Crescente\n");
45
46    return 0;
47  }
```

12.6 Repetição Interrompida Condicionada

Existem situações em que precisamos verificar uma propriedade de uma sequência. Para isto, fazemos uso de um indicador de passagem.

Agora, pode acontecer que, no momento em que o indicador de passagem recebe outro valor (diferente do valor inicial), não é mais necessário testar os outros números da sequência. Neste caso, podemos interromper a repetição.

Por exemplo, considere o problema dos dígitos adjacentes. Considere `n=12345678990`.
Lembre-se que a sequência gerada é composta pelos dígitos de `n` de trás para frente. A sequência gerada é então: $0, 9, 9, 8, 7, 6, 5, 4, 3, 2, 1$.
Quando o programa encontrar os dígitos adjacentes `9` e `9`, não é mais necessário verificar o restante da sequência.

Nestes casos, podemos utilizar o seguinte padrão de programação que usa o operador lógico `&&` e o indicador de passagem dentro da condição da repetição

A primeira condição `<condição da repetição>` garante a geração e/ou leitura da sequência. A segunda condição
`indicador == <valor inicial>`
garante que quando o indicador de passagem trocar de valor, a repetição é interrompida no momento da verificação da condição
`<condição da repetição> && indicador == <valor inicial>`,
uma vez que estamos usando o operador lógico `&&` e a condição `indicador == <valor inicial>` é falsa.

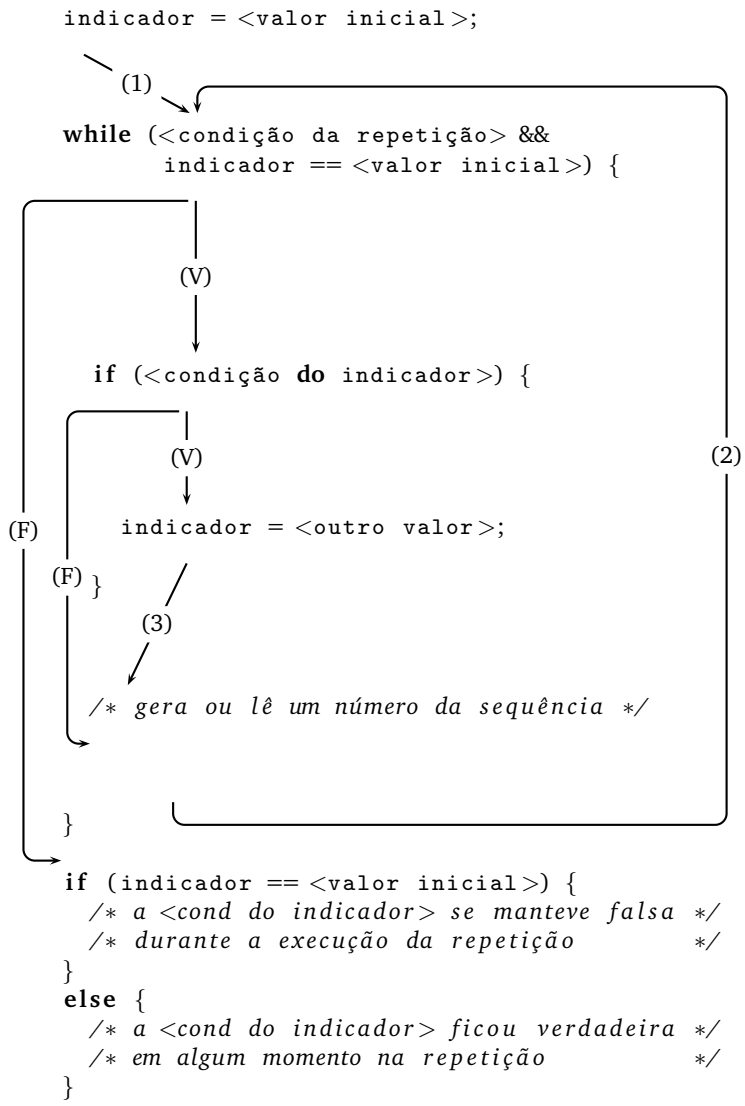


Figura 12: Padrão Repetição Interrompida Condicionada

12.7 Exercícios Recomendados

1. Dado um inteiro `p>1`, verificar se `p` é primo.

   Use indicador de passagem e o padrão repetição interrompida condicionada.

13 Números Reais - Tipo float

Ronaldo F. Hashimoto e Carlos H. Morimoto

Até o momento nos limitamos ao uso do tipo inteiro para variáveis e expressões aritméticas. Vamos introduzir agora o tipo real.

Ao final dessa aula você deverá saber:

- Declarar, ler e imprimir variáveis do tipo `float`.
- Calcular o valor de expressões aritméticas envolvendo reais.
- Utilizar variáveis reais em programas.

Para entender como são representados variáveis do tipo real, vamos falar um pouco sobre como os números inteiros e reais são representados no computador.

13.1 Representação de números inteiros

Os circuitos eletrônicos do computador armazenam a informação na forma binária (também chamada de digital). Um dígito binário pode assumir apenas 2 valores, representados pelos símbolos 0 (zero) e 1 (um), e que nos circuitos podem corresponder, por exemplo, a uma chave aberta/fechada, a um capacitor carregado/descarregado, etc. Esse elemento básico é conhecido como **bit**.

Os bits (dígitos binários) podem ser combinados para representar números da mesma forma que os dígitos decimais (dígitos de zero a nove), através de uma notação posicional, ou seja, o número 12 na base decimal equivale ao resultado da expressão $\boxed{1} \times 10^1 + \boxed{2} \times 10^0$. Essa mesma quantia pode ser representada por 1100 na base binária pois equivale ao resultado da expressão $\boxed{1} \times 2^3 + \boxed{1} \times 2^2 + \boxed{0} \times 2^1 + \boxed{0} \times 2^0$.

Por razões históricas, a memória do computador é dividida em bytes (conjunto de 8 bits), por isso a memória do seu computador pode ter, por exemplo, 128MB (mega bytes, onde 1MB = 2^{20} bytes) e o disco rígido 40GB (giga bytes, onde 1GB = 2^{30} bytes). Com um byte, é possível representar $2^8 = 256$ valores (todas as possíveis configurações de 8 bits de 00000000 a 1111111). Então os números decimais inteiros com 8 bits possíveis estão compreendidos de

$$\boxed{0} \times 2^7 + \boxed{0} \times 2^6 + \boxed{0} \times 2^5 + \boxed{0} \times 2^4 + \boxed{0} \times 2^3 + \boxed{0} \times 2^2 + \boxed{0} \times 2^1 + \boxed{0} \times 2^0 = 0$$

a

$$\boxed{1} \times 2^7 + \boxed{1} \times 2^6 + \boxed{1} \times 2^5 + \boxed{0} \times 2^4 + \boxed{1} \times 2^3 + \boxed{1} \times 2^2 + \boxed{1} \times 2^1 + \boxed{1} \times 2^0 = 255$$

ou seja, com 8 bits podemos representar inteiros de 0 a 255 (256 valores).

Nos computadores digitais, para representar de números negativos, é comum usar um bit para sinal. Se o bit de sinal é 0 (zero), então o número é positivo; caso contrário, o número é negativo. O bit de sinal é o bit mais à esquerda possível. Assim, o maior inteiro positivo com 8 bits é 01111111 , ou seja,

$$\boxed{0} \times 2^7 + \boxed{1} \times 2^6 + \boxed{1} \times 2^5 + \boxed{0} \times 2^4 + \boxed{1} \times 2^3 + \boxed{1} \times 2^2 + \boxed{1} \times 2^1 + \boxed{1} \times 2^0 = 127$$

A representação de números negativos nos computadores digitais é uma questão à parte que não vamos detalhar nesta aula. Só tenha em mente que o bit mais à esquerda deve ser 1 para números negativos. Um exemplo: a representação do -1 em 8 bits é 11111111.

Considerando um byte com o bit de sinal é possível representar então $2^8 = 256$ valores (de -128 a $+127$). Com 16 bits ou 2 bytes é possível representar 2^{16} valores (de -32768 a $+32767$) e, com uma palavra (conjunto de bits) de 32 bits, 2^{32} (de -2147483648 a $+2147483647$). Atualmente, boa parte dos computadores pessoais trabalham com palavras de 32 bits (embora já seja comum encontrar máquinas de 64 bits).

Na linguagem C, ao declararmos uma variável, o compilador reserva na memória o espaço necessário para representá-la. Como esse espaço é fixo (por exemplo, 32 bits para variáveis inteiras), é possível que durante uma computação o número de bits utilizado não seja suficiente para representar os valores necessários, e nesse caso, os resultados são, obviamente, inválidos.

Dependendo do maior/menor número que seu programa precisa representar, além de **int** você pode declarar variáveis inteiras como **char** (para usar palavras de 8 bits) – veja a aula 20 sobre caracteres.

13.2 Representação de Números Reais

Uma variável do tipo real é uma variável que pode conter números nos quais existe dígitos significativos à direita do ponto decimal. Por exemplo, 3.2, 21.43 0.12, etc.

Na memória do computador não podemos armazenar $1/2$ bit (apenas os zeros e uns). Como então representar um número fracionário, ou real? A representação é análoga à notação científica, feita em ponto flutuante da seguinte forma:

$$0.x_1x_2x_3...x_k \times B^e$$

onde $x_1x_2x_3...x_k$ é a mantissa (os k dígitos mais significativos do número), B é a base e e é o expoente (através do qual se determina a posição correta do dígito mais significativo do número em ponto flutuante). Essa notação permite a representação de uma faixa bastante grande de números, como por exemplo:

Número	Notação Científica	Mantissa	Base	Expoente
1000000000	0.1×10^{10} ou 1E9	1	10	10
123000	0.123×10^6 ou 1.23E5	123	10	6
456.78	0.45678×10^3 ou 4.5678E2	45678	10	3
0.00123	0.123×10^{-2} ou 1.23E$-$3	123	10	-2

Note que o "ponto flutuante" corresponde à posição do ponto decimal, que é "ajustado" pelo valor do expoente, e que nesse exemplo a mantissa, a base e o expoente são agora números inteiros. Uma notação semelhante pode ser utilizada para números binários, e reservamos espaço na memória (ou bits de uma palavra) para armazenar a mantissa e o expoente (a base pode ser pré-determinada, 2 no caso dos computadores digitais). Assim, a representação de um número real com 32 bits poderia usar 24 bits para a mantissa e 7 para o expoente. Como você mesmo pode observar, da mesma forma que os inteiros, os números em ponto flutuante são armazenados como um conjunto fixo de bytes, de modo que a sua precisão é limitada.

Dessa forma, o computador é uma máquina com capacidade de armazenamento finita. Assim, o conjunto de números que podem ser representados no tipo real não é o mesmo conjunto de números reais da matemática, e sim um subconjunto dos números racionais.

13.3 Variável Tipo Real

Os tipos de dados inteiros servem muito bem para a maioria dos programas, contudo alguns programas orientados para aplicações matemáticas frequentemente fazem uso de **números reais** (ou em **ponto flutuante**). Para este tipo de dados, em C, podemos utilizar os tipos **float** e **double**.

A diferença entre estes dois tipos é que no tipo de dado **double**, podemos representar uma quantidade maior de números reais que no tipo **float**. O tipo **double** usa 8 bytes para guardar um número em ponto flutuante (53

bits para a mantissa e 11 para o expoente); enquanto o **float** usa 4 bytes (24 bits para a mantissa e 8 para o expoente).

Os valores do tipo **float** são números que podem, em valor absoluto, serem tão grandes com 3.4×10^{38} ou tão pequenos quanto 3.4×10^{-38}. O tamanho da mantissa para este tipo de dado é 7 dígitos decimais e são necessários 4 bytes de memória para armazenar um valor deste tipo.

Os valores **double** são números que podem, em valor absoluto, serem tão grandes com 1.7×10^{308} ou tão pequenos quanto 1.7×10^{-308}. O tamanho da mantissa para este tipo de dado é 15 dígitos decimais e são necessários 8 bytes de memória para armazenar um valor deste tipo.

Assim, o tipo **float** tem uma precisão de 6 a 7 casas decimais com o expoente variando entre 10^{-37} a 10^{+38} e o tipo **double** uma precisão de 15 casas decimais com expoente variando entre 10^{-308} a 10^{+308} ocupando um espaço maior para armazenar um valor na memória. Isto significa que um número como 123456.78901234 será armazenado apenas como $1.234567E6$ em uma variável do tipo **float** ficando o restante além da precisão possível para a representação.

Neste curso, vamos usar o tipo **float**.

A forma para declarar uma variável do tipo **float** é a mesma para declarar variáveis do tipo **int**; só que em vez de usar a palavra chave **int**, deve-se usar a palavra **float**:

float `<nome_da_variavel>;`

Exemplo: declaração de uma variável do tipo **float** de nome `"r"`

float `r;`

Se você quiser declarar várias variáveis, é possível fazer da seguinte forma:

float `<nome_da_variavel_1>, <nome_da_variavel_2>, <nome_da_variavel_3>,..., <nome_da_variavel_n>;`

Exemplo: declaração de duas variáveis do tipo **float** `"r1"` e `"r2"`.

float `r1, r2;`

13.4 Leitura de um Número Real pelo Teclado

Como vimos nas aulas passadas, para ler um número inteiro pelo teclado, nós usamos o "`%d`" dentro do comando `scanf`. Assim, para ler um inteiro `x` fazemos:

```
1       int x;
2
3       printf ("Entre com um numero inteiro x > 0: ");
4       scanf  ("%d", &x);
```

Para ler um número real pelo teclado, você deve utilizar "`%f`" dentro do comando `scanf`.

Para mostrar um exemplo, considere o seguinte trecho de programa que lê um número real:

```
1       float x;
2
3       printf ("Entre com um número real: ");
4       scanf  ("%f", &x);
```

13.5 Impressão de Números Reais

Como vimos nas aulas passadas, para imprimir um número inteiro na tela, nós usamos o "%d" dentro do comando `printf`. Assim, para imprimir um inteiro `x` fazemos:

```
1       int x;
2
3       printf ("Entre com um numero x > 0: ");
4       scanf  ("%d", &x);
5
6       printf ("Número lido foi = %d\n", x);
```

Para imprimir um número real na tela, nós podemos usar o "%f" dentro do comando `printf`:

```
1       float x;
2
3       printf ("Entre com um número real: ");
4       scanf  ("%f", &x);
5
6       printf ("Numero Digitado = %f\n", x);
```

É possível imprimir números reais ainda de outras formas:

%e	imprime um valor real em notação científica
%f	imprime um valor real em notação decimal
%g	imprime um valor real na notação científica ou decimal, como for mais apropriada

Veja o seguinte exemplo:

```
1       #include <stdio.h>
2
3       int main () {
4         float  f = 3.141592654;
5
6         printf("formato e: f=%e\n", f);
7         printf("formato f: f=%f\n", f);
8         printf("formato g: f=%g\n", f);
9
10        return 0;
11      }
```

A saída desse programa é:

```
formato e: f=3.141593e+000
formato f: f=3.141593
formato g: f=3.14159
```

13.5.1 Formatação de Impressão de Números Reais

Muitas vezes, para facilitar a visualização dos resultados, é necessário formatar os dados na saída do programa. Tanto o formato `%d` quanto o `%f` podem ser formatados no sentido de reservar um número de dígitos para impressão. Para usar formatação, você pode colocar entre o `%` e o caractere definindo o tipo (`d` ou `f`) o seguinte:

- **um sinal de menos**: especifica ajustamento à esquerda (o normal é à direita).
- **um número inteiro**: especifica o tamanho mínimo do campo. Se o número a ser impresso ocupar menos espaço, o espaço restante é preenchido com brancos para completar o tamanho desejado, mas se o número ocupar um espaço maior, o limite definido não é respeitado.
- **um ponto seguido de um número**: especifica o tamanho máximo de casas decimais a serem impressos após o ponto decimal. A precisão padrão para números reais é de 6 casas decimais.

Exemplos:

Considere a variável real `cempi = 314.159542` e veja como ela pode ser impressa usando diferentes formatos (as barras verticais facilitam a visualização):

```
1       float cempi = 314.159542;
2
3       printf("cempi = |%-8.2f|\n", cempi);
4       printf("cempi = |%8.2f|\n", cempi);
5       printf("cempi = |%8.4f|\n", cempi);
6       printf("cempi = |%8.4f|\n", cempi * 1000);
```

A impressão na tela fica:

```
cempi = |314.16  |
cempi = |  314.16|
cempi = |314.1595|
cempi = |314159.5313|
```

Observe que 8 casas incluem o ponto decimal, e são suficientes para os primeiros 3 `printf`'s. No último `printf` esse limite não é obedecido, pois o número a ser impresso ocupa um lugar maior que 8. Observe também que o tipo **float** perde precisão em torno da sexta casa decimal, daí os últimos digitos de `cempi * 1000` não estarem corretos.

13.6 Escrita de Números Reais

Números em ponto flutuante podem ser definidos de diversas formas. A mais geral é uma série de dígitos com sinal, incluindo um ponto decimal, depois um 'e' ou 'E' seguido do valor do expoente (a potência de dez) com sinal. Por exemplo: `-1.609E-19` e `+6.03e+23`. Essas cIonstantes podem ser utilizadas em expressões como por exemplo:

```
1       float x = 3.141595426;
2       float y = 1.23e-23;
3 I
```

Na definição de números reais pode-se omitir sinais positivos, a parte de expoente e a parte inteira ou fracionária. Exemplos:

- 3.14159
- .2
- 4e16
- .8e−5
- 100

Não se deve usar espaços dentro de um número em ponto flutuante: O número `3.34␣E+12` está errado.

13.7 Expressões Aritméticas Envolvendo Reais

Ao utilizarmos números reais em nossos programas, é comum misturar números e variáveis inteiras com reais em nossas expressões aritméticas. Para cada operador (+, -, *, /, etc) da expressão, o compilador precisa decidir se a operação deve ser realizada como inteira ou como real, pois como a forma de representação de inteiros e reais é diferente, as operações precisam ser feitas usando a mesma representação. A regra básica é, se os operandos tiverem tipos diferentes, a operação é realizada usando o "maior" tipo, ou seja, se um dos operandos for real, o resultado da operação é real, caso contrário, a operação é inteira.

13.7.1 Observação quanto à Divisão

```
1       int i,j;
2       float y;
3
4       i = 5 / 3; /* divisão inteira e o resultado é 1 (5 e 3 são inteiros) */
5       y = 5 / 3; /* divisão inteira e o resultado é 2.0 (y é real) */
6       y = 5.0 / 2; /* divisão tem como resultado 2.5 (o numerador é real) */
7       y = 5 / 2.0; /* divisão tem como resultado 2.5 (o denominador é real) */
8
9       y = i / 2; /* divisão inteira (i e 2 são inteiros) */
10      y = i / 2.0; /* divisão em ponto flutuante (denominador real) */
11      y = i / j; /* divisão inteira (i e j são inteiros) */
12      y = (1.0 * i) / j; /* divisão em ponto flutuante (numerador real) */
13      y = 1.0 * (i / j); /* divisão inteira (i e j são inteiros) */
14      i = y / 2; /* parte inteira da divisão em i (divisão real, mas i é inteiro) */
15 I
```

Veja a saída do programa abaixo e tente entender o que acontence no primeiro e no segundo printf:

```
1       #include <stdio.h>
2
3       int main () {
4         int i=4;
5         int j=5;
6         int k;
7         float f = 5.0;
8         float g;
9
10        k = 6*(j/i); /* variável inteira k recebe resultado de expressão inteira */
11        g = 6*(f/i); /* variável real g recebe resultado de expressão real */
12        printf("1: k=%d g=%f\n", k, g);
13
14        g = 6*(j/i); /* variável real g recebe resultado de expressão inteira */
15        k = 6*(f/i); /* variável inteira k recebe resultado de expressão real */
16        printf("2: k=%d g=%f\n", k, g);
17
18        return 0;
19      }
20 I
```

A saída dos `printf`'s é:

```
1: k=6 g=7.500000
2: k=7 g=6.000000
```

Lembre-se que em uma atribuição, cada expressão é calculada (lado direito) e o resultado é depois armazenado na variável correspondente, definida no lado esquerdo da atribuição. Nas atribuições antes do primeiro `printf`,

o tipo da expressão é o mesmo da variável, mas nas atribuições seguintes, os tipos são diferentes. Observe portanto que o tipo da variável que recebe a atribuição NÃO influencia a forma de calcular as expressões. Após o cálculo, o resultado é convertido ao tipo da variável (ou seja, inteiro 6 passa a real 6.0 e real 7.5 passa a inteiro 7). É possível forçar a mudança de tipos de um termo dentro de expressão através de definições explícitas conhecidas como **type casting**. Observe o exemplo abaixo:

```
1       #include <stdio.h>
2
3       int main () {
4         int i=4;
5         int j=5;
6         int k;
7         float f = 5.0;
8         float g;
9
10        /* variável inteira k recebe resultado de expressão inteira */
11        k = 6*(j/i);
12
13        /* variável real g recebe resultado de expressão inteira, */
14        /* pois a variável f foi explicitamente convertida para o tipo int */
15        g = 6*((int)f/i);
16
17        printf("1: k=%d g=%f\n", k, g);
18
19
20        /* o número 6 é promovido a float, e portanto o resultado é real */
21        /* uma forma mais simples seria definir o número 6 como 6.0 */
22        g = (float)6*j/i;
23
24
25        /* variável inteira k recebe a parte inteira do resultado da expressão real */
26        k = 6*(f/i);
27
28        printf("2: k=%d g=%f\n", k, g);
29
30        return 0;
31      }
32 I
```

13.8 Exercício

Dado um natural n, determine o número harmônico H_n definido por

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

Solução Comentada:

A somatória indica que precisamos realizar as seguites operações:

$$H_n = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}$$

Imediatamente, já podemos observar que precisamos fazer n somas, gerando uma sequência de inteiros de 1 a n. Para isso, precisamos de uma repetição que tem a seguinte estrutura:

```
1       i = 1;
2       while (i<=n) {
3         soma = soma + 1 / i;
4         i = i + 1;
5       }
6 I
```

ou usando a forma mais sucinta com o comando **for**:

```
1       for (i=1; i<=n; i++) {
2         soma = soma + 1 / i;
3       }
4 I
```

Observe que i pode ser uma variável inteira e soma PRECISA ser uma variável real. Por que não utilizamos então todas as variáveis reais? Por várias razões. Uma delas é a consistência, já que o número de termos da soma é inteiro, faz sentido (fica mais fácil de entender) se a variável for inteira, e devido ao desempenho do seu programa, pois as operações com inteiros são realizadas mais rapidamente pelo computador que as operações em ponto flutuante (real).

A solução final para esse programa seria:

```
1       #include <stdio.h>
2       int main () {
3
4         float soma = 0;
5         int i;
6
7         printf("Entre com o valor de n>0: ");
8         scanf("%d", &n);
9
10        for (i = 1; i<=n; i++) {
11          soma = soma + 1 / i;
12        }
13
14        printf("o número harmonico H%d = %f\n", n, soma);
15
16        return 0;
17      }
18 I
```

Aparentemente, essa solução está correta, porém, ela possui um erro difícil de notar. Teste esse programa, exatamente como está escrito acima, e verifique que a saída do programa é sempre 1, para qualquer $n > 0$. Por que?

Vimos que o compilador decide, para cada operação, se ela deve ser realizada como inteiro e como real, dependendo do tipo dos operandos envolvidos. Veja com atenção a linha:

```
soma = soma + 1 / i;
```

Devido a precedência dos operadores, a divisão é feita antes da soma. Como os operandos 1 e i são ambos inteiros, o resultado da divisão também é inteiro, ou seja, quando $i > 1$, o resultado é sempre 0, daí o resultado ao final sempre ser 1. Para resolver esse erro, basta explicitamente colocar a constante real 1.0 ou colocar um casting antes do número inteiro 1 ou na variável inteira i, como abaixo:

`soma = soma + 1.0 / i;` | `soma = soma + (float) 1 / i;` | `soma = soma + 1 / (float) i;`

Uma solução final para esse programa seria:

```
1       #include <stdio.h>
2       int main () {
3
4         float soma = 0;
5         int i;
6
7         printf("Entre com o valor de n>0: ");
8         scanf("%d", &n);
9
10        for (i = 1; i<=n; i++) {
11          soma = soma + 1.0 / i;
12        }
13
14        printf("o número harmonico H%d = %f\n", n, soma);
15
16        return 0;
17      }
18 I
```

13.9 Exercícios recomendados

1. Dado um número inteiro $n > 0$, calcular o valor da soma

$$s_n = 1/n + 2/(n-1) + 3/(n-2) + 3/(n-2) + \ldots + n/1.$$

2. Dado um número real x e um número real $epsilon > 0$, calcular uma aproximação de e^x através da seguinte série infinita:

$$e^x = 1 + x + x^2/2! + x^3/3! + \ldots + x^k/k! + \ldots$$

   Inclua na aproximação todos os termos até o primeiro de valor absoluto (módulo) menor do que $epsilon$.

14 Fórmula de Recorrência e Séries (Somas Infinitas)

Ronaldo F. Hashimoto e Carlos H. Morimoto

Nessa aula vamos introduzir fórmulas de recorrência e o uso das mesmas para o cálculo de séries (somas infinitas).

Ao final dessa aula você deverá saber:

- Descrever o que são fórmulas de recorrência.
- Descrever o que são erro absoluto e relativo.
- Escrever programas em C a partir de fórmulas de recorrência.

14.1 Fórmula de Recorrência

Uma fórmula de recorrência é uma relação entre os termos sucessivos de uma sequência numérica. Dessa forma, usando uma fórmula de recorrência, é possível obter o próximo termo da sequência usando o valor de termos anteriores. Um exemplo clássico é a sequência de Fibonacci definida pela fórmula de recorrência

$$\begin{cases} F_1 = 1 \\ F_2 = 1 \\ F_i = F_{i-1} + F_{i-2} \quad \text{para } \ i \geq 3. \end{cases}$$

Note que para determinar o termo F_i é necessário ter os dois termos anteriores F_{i-1} e F_{i-2}.

14.2 Exercício de Fórmula de Recorrência: Raiz Quadrada

Dados $x \geq 0$ e eps, $0 < eps < 1$, números reais, calcular uma aproximação para raiz quadrada de x através da sequência de números gerada pela seguinte fórmula de recorrência:

$$\begin{cases} r_0 = x \\ r_{k+1} = (r_k + x/r_k)/2 \quad \text{para } \ k > 0. \end{cases}$$

Gere a sequência até um k tal que $|r_k - r_{k-1}| < eps$. A raiz quadrada de x é o último valor da sequência, isto é, r_k.

Note que de uma certa maneira o real eps controla a precisão da raiz quadrada de x.

Solução:

Gerar os números da sequência $r_0, r_1, r_2, \ldots, r_k$ usando uma repetição.

```
1       float r, x, erro, eps;
2
3       r = x; erro = eps;
4       while (erro >= eps) {
5         r = (r + x / r) / 2;
6         printf ("r = %f\n", r);
7
8         /* atualiza erro */
9         ...
10      }
```

A repetição acima de fato imprime cada elemento r_k da sequência. No entanto, para calcular o `erro`=$|r_k - r_{k-1}|$ é necessário guardar o termo anterior. Assim, vamos declarar mais uma variável `rant` e fazer a geração da sequência da seguinte forma:

```
1       float r, rant, x, erro, eps;
2
3       rant = r = x; erro = eps;
4       while (erro >= eps) {
5         r = (rant + x / rant) / 2;
6         printf ("r = %f\n", r);
7
8         /* atualiza erro */
9         erro = r - rant;
10
11        /* atualiza rant */
12        rant = r;
13      }
```

Agora, note que o cálculo do erro no trecho de programa acima está errado, uma vez que `erro`=$|r_k - r_{k-1}|$ (valor absoluto). Para consertar isso, temos que verificar se `erro` ficou negativo. Em caso afirmativo, devemos trocar o sinal de `erro`:

```
1       float r, rant, x, erro, eps;
2
3       rant = r = x; erro = eps;
4       while (erro >= eps) {
5         r = (rant + x / rant) / 2;
6         printf ("r = %f\n", r);
7
8         /* atualiza erro */
9         erro = r - rant;
10        if (erro < 0)
11          erro = -erro;
12
13        /* atualiza rant */
14        rant = r;
15      }
```

Note ainda que devemos garantir que o programa funcione para `x=0`. Como a raiz quadrada de zero é zero, podemos fazer com que quando `x=0`, o programa não entre no laço colocando uma condição `(erro>= eps && x>0)`.

Assim, a solução final do exercício é:

```c
#include <stdio.h>

int main () {
  float r, rant, x, erro, eps;

  printf ("Entre com x >= 0: ");
  scanf ("%f", &x);

  printf ("Entre com 0 < eps < 1: ");
  scanf ("%f", &eps);

  rant = r = x; erro = eps;
  while (erro >= eps && x>0) {
    r = (rant + x / rant) / 2;

    /* atualiza erro */
    erro = r - rant;
    if (erro < 0)
      erro = -erro;

    /* atualiza rant */
    rant = r;
  }

  printf ("Raiz de %f = %f\n", x, r);

  return 0;
}
```

14.3 Erro Absoluto e Erro Relativo

Dado um número x e uma aproximação y para x, o erro (também chamado de erro absoluto) da aproximação y em relação x é definido como $|y - x|$. Quando a grandeza de x não é próxima da de 1, o erro absoluto pode não ser a maneira mais adequada de medir a qualidade da aproximação y. Por exemplo, os erros absolutos de 1.01 em relação a 1.00 e de 0.02 em relação a 0.01 são idênticos, mas é claro que a primeira aproximação é muito melhor que a segunda.

Face à limitada avaliação de uma aproximação conferida pelo erro absoluto, tenta-se definir o erro relativo a y em relação a x como sendo

$$|\frac{y-x}{x}|$$

Assim, nos dois exemplos anteriores, os erros relativos são respectivamente de 0.01 (ou 1%) e 1.00 (ou 100%). Contudo esta definição é incompleta quando $x = 0$. Neste caso, a divisão por 0 não pode ser realizada e adotam-se valores arbitrários para o erro relativo. No caso de também ocorrer que $y = 0$, a aproximação certamente é perfeita e adota-se que o erro é 0. No caso de $y \neq 0$, a aproximação é certamente insatisfatória e adota-se o valor arbitrário 1 para o erro relativo. Assim, definimos

$$errorel(y, x) = \begin{cases} |(y-x)/x| & \text{se } x \neq 0 \\ 0 & \text{se } x = 0 = y \\ 1 & \text{se } x = 0 \neq y \end{cases}$$

14.4 Exercício da Raiz Quadrada com Erro Relativo

Resolver o exercício da raiz quadrada usando erro relativo em vez de erro absoluto, ou seja, gerar a sequência até um k tal que $errorel(r_k, r_{k-1}) < eps$.

Solução:

A única diferença deste exercício com relação ao anterior é o cálculo do erro. Esse cálculo pode ser feito da seguinte forma:

```
float r, rant, x, erro;

if (rant != 0) {
  erro = (r - rant) / rant;
  if (erro < 0)
    erro = -erro;
}
else { /* rant == 0 */
  if (r == 0)
    erro = 0;
  else
    erro = 1;
}
```

Assim, a solução final do exercício é:

```
# include <stdio.h>

int main () {
  float r, rant, x, erro;

  printf ("Entre com x >= 0: ");
  scanf ("%f", &x);

  printf ("Entre com 0 < eps < 1: ");
  scanf ("%f", &eps);

  erro = r = x;
  while (erro >= eps) {
    r = (rant + x / rant) / 2;

    /* atualiza erro */
    if (rant != 0) {
      erro = (r - rant) / rant;
      if (erro < 0)
        erro = -erro;
    }
    else { /* rant == 0 */
      if (r == 0)
        erro = 0;
      else
        erro = 1;
    }

    /* atualiza rant */
    rant = r;
  }

  printf ("Raiz de %f = %f\n", x, r);

  return 0;
}
```

14.5 Exercício de Cálculo de Séries

Dados x e eps, $0 < eps < 1$, reais, obter uma aproximação da série

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^k}{k!} + \cdots$$

com precisão eps, isto é, somar os termos da série até aparecer um termo cujo valor absoluto seja menor que eps.

Primeiramente, vamos mostrar uma forma que você não deve usar para resolver este exercício.

- $\left|\frac{x^k}{k!}\right|$ tende a zero quando k tende a $+\infty$.
- Usando um comando de repetição, gerar uma sequência de números $k = 1, 2, 3, 4, \ldots,$
- Calcular $p = x^k$.
- Calcular $fat = k!$.
- Calcular $t = p/fat$.
- Acumular t em uma variável $soma$.
- Repetir estes cálculos até um k tal que $\left|\frac{x^k}{k!}\right| < eps$.

Esta solução poderia ser escrita como:

```
1       float soma, t, eps, x, pot, fat;
2       int k;
3
4       soma = 1; t = 1;  k = 1;
5       while (|t| >= eps) {
6         /* calcule pot = x^k; */
7         /* calcule fat = k!; */
8         t = pot / fat;
9         soma = soma + t;
10        k++;
11      }
```

Para o cálculo de pot e fat, é possível aproveitar os valores de pot e fat anteriores da seguinte forma:

```
1       float soma, t, eps, x, pot, fat;
2       int k;
3
4       soma = t = k = 1;
5       pot = fat = 1;
6       while (|t| >= eps) {
7         /* calcule pot = x^k; */
8         pot = pot * x;
9         /* calcule fat = k!; */
10        fat = fat * k;
11        t = pot / fat;
12        soma = soma + t;
13        k++;
14      }
```

Esta solução é ruim, pois como não sabemos até que valor k vai assumir, a variável fat que recebe o fatorial de k, pode estourar facilmente, mesmo fat sendo uma variável do tipo `float`.

Assim a solução acima não é considerada uma boa solução. Uma boa solução não deve envolver o cálculo do fatorial.

A idéia é calcular um termo da série usando o termo anterior. Observe que

$$t_{k-1} = \frac{x^{k-1}}{(k-1)!}$$

O próximo termo t_k é

$$t_k = \frac{x^k}{k!} = \frac{x^{k-1} \times x}{(k-1)! \times k} = \frac{x^{k-1}}{(k-1)!} \times \frac{x}{k} = t_{k-1} \times \frac{x}{k}$$

Assim, para calcular o próximo termo da série, basta multiplicar o termo anterio pelo fator $\frac{x}{k}$.

Assim, uma melhor solução seria:

```
1       float soma, t, eps, x;
2       int k;
3
4       soma = t = k = 1;
5       while (|t| >= eps) {
6         t = t * x / k;
7         soma = soma + t;
8         k++;
9       }
```

Note que esta solução não envolve diretamente o cálculo de fatorial. A solução completa seria:

Solução:

```
1   # include <stdio.h>
2
3   int main () {
4
5     float soma, t, eps, x, abs_t;
6     int k;
7
8     printf ("Entre com x: ");
9     scanf ("%f", &x);
10
11    printf ("Entre com 0 < eps < 1: ");
12    scanf ("%f", &eps);
13
14    soma = abs_t = t = k = 1;
15
16    while (abs_t >= eps) {
17      t = t * x / k;
18      soma = soma + t;
19      k++;
20      abs_t = t;
21      if (abs_t < 0)
22        abs_t = -abs_t;
23    }
24
25    printf ("exp(%f) = %f\n", x, soma);
26
27    return 0;
28  }
```

Note o `if` no final da repetição (linha 21) para calcular o módulo do termo t.

14.6 Outro Exercício de Cálculo de Séries

Dados x e ϵ reais, $\epsilon > 0$, calcular uma aproximação para sen x através da seguinte série infinita

$$\text{sen } x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + (-1)^k \frac{x^{2k+1}}{(2k+1)!} + \ldots$$

incluindo todos os termos até que $\frac{|x^{2k+1}|}{(2k+1)!} < \epsilon$.

Solução:

Neste exercício, temos que calcular o termo seguinte em função do termo anterior. Assim, o termo anterior é

$$t_{k-1} = (-1)^{k-1} \cdot \frac{x^{2(k-1)+1}}{(2(k-1)+1)!} = (-1)^{k-1} \cdot \frac{x^{2k-1}}{(2k-1)!}$$

$$\begin{aligned} t_k &= (-1)^k \cdot \frac{x^{2k+1}}{(2k+1)!} = (-1)^{k-1} \cdot (-1) \cdot \frac{x^{(2k-1)+2}}{(2k-1)! \cdot (2k) \cdot (2(k+1))} = \\ &= (-1)^{k-1} \cdot \frac{x^{2k-1}}{(2k-1)!} \cdot \frac{-x^2}{(2k) \cdot (2(k+1))} = \\ &= t_{k-1} \cdot \frac{-x^2}{(2k) \cdot (2(k+1))} \end{aligned}$$

Assim, neste exercício, podemos calcular o termo seguinte em função do termo anterior apenas multiplicando-o pelo fator $\frac{-x^2}{(2k) \cdot (2(k+1))}$. Um esboço de uma solução seria:

- Usando um comando de repetição, gerar uma sequência de números $k = 1, 2, 3, 4, \ldots,$
- Calcular $t_k = t_{k-1} \times (-x^2)/\left((2k) \cdot (2(k+1))\right)$.
- Acumular t em uma variável $soma$.
- Repetir estes cálculos até um k tal que $|t_k| < \epsilon$.

Note um par de parênteses a mais no divisor do fator multiplicativo. Este par de parênteses é necessário para fazer a divisão corretamente. Uma solução completa seria:

Solução:

```
1   # include <stdio.h>
2
3   int main () {
4
5     float soma, t, eps, x, abs_t;
6     int k;
7
8     printf ("Entre com x: ");
9     scanf ("%f", &x);
10
11    printf ("Entre com 0 < eps < 1: ");
12    scanf ("%f", &eps);
13
14    soma = abs_t = t = x;
15    if (abs_t < 0) abs_t = -abs_t;
16    k = 1;
17
18    while (abs_t >= eps) {
19      t = - t * x * x / ((2*k)*(2*k+1));
20      soma = soma + t;
21      k++;
22      abs_t = t;
23      if (abs_t < 0)
24        abs_t = -abs_t;
25    }
26
27    printf ("sen(%f) = %f\n", x, soma);
28
29    return 0;
30  }
```

14.7 Exercícios Recomendados

Dados x real e N natural, calcular uma aproximação para $\cos x$ através dos N primeiros termos da seguinte série:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + (-1)^k \frac{x^{2k}}{(2k)!} + \ldots$$

15 Funções - Introdução

Ronaldo F. Hashimoto e Carlos H. Morimoto

O objetivo desta aula é introduzir o conceito de função e sua utilidade. Ao final dessa aula você deverá saber:

- Justificar as vantagens do uso de funções em programas.
- Identificar em seus programas oportunidades para criação e uso de funções.
- Definir os parâmetros que a função precisa receber (parâmetros de entrada) e o que a função precisa devolver (o tipo do valor devolvido pela função).

15.1 Exercício de Aquecimento

Faça um programa que leia um real x e um inteiro $n > 0$ e calcule x^n.

Uma solução deste exercício poderia usar a seguinte estratégia:

- Gerar uma sequência de n números reais $x, x, x, \ldots, x$.
- Para cada número gerado, acumular o produto em uma variável pot:

$$pot = pot * x$$

Um trecho de programa para esta solução poderia ser:

```
1       float pot, x;
2       int cont, n;
3       cont = 0; pot = 1;
4       while (cont < n) {
5         pot = pot * x;
6         cont = cont + 1;
7       }
8       /* no final pot contém x^n */
```

15.2 Exercício de Motivação

Faça um programa que leia dois números reais x e y e dois números inteiros $a > 0$ e $b > 0$, e calcule o valor da expressão $x^a + y^b + (x - y)^{a+b}$.

Este programa deve calcular a potenciação de dois números três vezes: uma para x^a, outra para y^b e outra para $(x - y)^{a+b}$.

Para calcular x^a, temos o seguinte trecho de programa:

```
1       float pot, x;
2       int cont, a;
3       cont = 0; pot = 1;
4       while (cont < a) {
5         pot = pot * x;
6         cont = cont + 1;
7       }
8       /* no final pot contem x^a */
```

Para calcular y^b, temos o seguinte trecho de programa:

```
1       float pot, y;
2       int cont, b;
3       cont = 0; pot = 1;
4       while (cont < b) {
5         pot = pot * y;
6         cont = cont + 1;
7       }
8       /* no final pot contém y^b */
```

Para calcular $(x-y)^{a+b}$, temos o seguinte trecho de programa:

```
1       float pot, x, y;
2       int cont, a, b;
3       cont = 0; pot = 1;
4       while (cont < a+b) {
5         pot = pot * (x-y);
6         cont = cont + 1;
7       }
8       /* no final pot contém (x-y)^(a+b) */
```

Tirando as variáveis que mudam, isto é, os expoentes (variáveis a, b e $a+b$) e as bases (variáveis x, y e $x-y$), praticamente temos o mesmo código. Na elaboração de programas, é comum necessitarmos escrever várias vezes uma mesma sequência de comandos, como no exemplo acima.

Assim, seria muito interessante termos uma maneira de escrevermos um código que poderia ser aproveitado, mudando somente os **valores** (note que está escrito **valores** e não nomes) das variáveis que são o expoente e a base da potenciação. Algo como:

```
1       float pot, base;
2       int cont, expoente;
3       cont = 0; pot = 1;
4       while (cont < expoente) {
5         pot = pot * base;
6         cont = cont + 1;
7       }
8       /* no final pot contém base elevado a expoente */
```

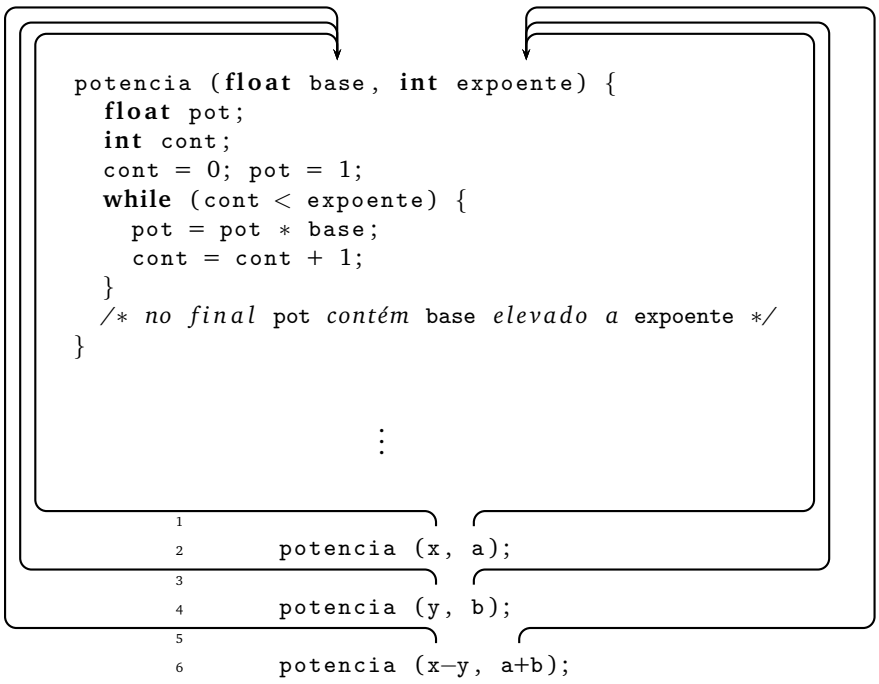
Assim, se quisermos calcular x^a, devemos fazer algo como `expoente ← a` e `base ← x` e rodarmos o código acima. Da mesma forma, para calcular y^b e $(x-y)^{a+b}$ devemos fazer, de alguma forma, `expoente ← b`, `base ← y` e `expoente ← a+b` e `base ← (x-y)`, respectivamente, e rodar o código acima.

A idéia aqui é ainda escrevermos este código somente uma vez! Para isso, vamos dar um nome para o nosso trecho de código. Que tal `potencia`? Assim, toda vez que quisermos utilizar o código acima, simplesmente usuaríamos seu nome que é `potencia` . Poderíamos fazer:

```
1     potencia (float base, int expoente) {
2       float pot;
3       int cont;
4       cont = 0; pot = 1;
5       while (cont < expoente) {
6         pot = pot * base;
7         cont = cont + 1;
8       }
9       /* no final pot contém base elevado a expoente */
10    }
```

O trecho de código acima de nome `potencia` na verdade é uma **função**. A idéia de função é fazer um subprograma de nome `potencia` que seja utilizado, para o exemplo acima, três vezes.

Note agora que as variáveis `base` e `expoente` estão declaradas entre parênteses e depois do nome `potencia`. Por que? A idéia para calcular x^a não é fazer algo como `expoente ← a` e `base ← x`? Então, para calcular x^a, a gente poderia colocar o conteúdo das variáveis `x` e `a` nas variáveis `base` e `expoente`, respectivamente, usando o comando `potencia (x, a)`. Observe no diagrama abaixo:



O comando `potencia (x, a)` coloca o conteúdo das variáveis `x` e `a` (seus valores) nas variáveis `base` e `expoente` respectivamente, como indicado pelas setas. Depois executa os comandos com os respectivos valores das variáveis `base` e `expoente` atualizados. No final da execução da função `potencia`, ou seja, depois que sair do laço, a variável `pot` contém x^a.

Da mesma forma, podemos então calcular y^b e $(x-y)^{a+b}$ usando o comando `potencia (y, b)` e o comando `potencia (x-y, a+b)`, respectivamente. Assim, ao final de cada execução da função `potencia`, a variável `pot` contém y^b e $(x-y)^{a+b}$, respectivamente.

Agora, você deve estar se perguntando o que fazer com cada resultado que está na variável `pot`? Aliás, como `pot` pode guardar três valores se `pot` é uma variável **float**. Na verdade, `pot` guarda somente **um** valor real. Toda vez que o código `potencia` é executado, o valor do último `pot` é perdido!

Ah! Então, para cada vez que o código de `potencia` for executado, a gente poderia imprimir o valor final da variável `pot`. Aí a gente tem o resultado de cada potenciação! Legal, mas, isto não resolveria o problema de calcular $x^a + y^b + (x-y)^{a+b}$. O que teríamos que fazer é guardar o resultado das potenciações em outras três variáveis, por exemplo, `potx`, `poty` e `potxy` e no final imprimir a soma `potx` + `poty` + `potxy`.

Para indicar que o resultado de `potencia (x, a);` será colocado na variável `potx`, vamos escrever o uso da função `potencia` como `potx = potencia (x, a);`. Da mesma forma, para as variáveis `poty` e `potxy`. Assim, teríamos:

```
1       potx = potencia (x, a);
2
3       poty = potencia (y, b);
4
5       potxy = potencia (x-y, a+b);
6
7       soma = potx + poty + potxy;
8       printf ("Resultado = %f\n", soma);
```

Assim, `potx = potencia (x, a);`, `poty = potencia (y, b);` e `potxy = potencia (x-y, a+b);` indicam que o valor da variável `pot` de cada execução da função `potencia` é colocado nas variáveis `potx`, `poty` e `potxy`, respectivamente. Depois é só imprimir a soma das variáveis `potx`, `poty` e `potxy`.

Agora, como fazer com que a função `potencia` coloque corretamente "para fora" o valor da variável `pot`? Nós vamos indicar isso colocando no final da função `potencia` o comando **return** `pot;`. Veja como fica então:

```
1       potencia (float base, int expoente) {
2         float pot;
3         int cont;
4         cont = 0; pot = 1;
5         while (cont < expoente) {
6           pot = pot * base;
7           cont = cont + 1;
8         }
9         /* no final pot contém base elevado a expoente */
10        return pot;
11      }
```

Assim, o uso `potx = potencia (x, a);` faz com que `expoente ← a` e `base ← x` e executa o código de `potencia`; no final da execução, a variável `pot` contém o valor de x^a; o comando **return** `pot;` coloca o valor que está na variável `pot` da função `potencia` em um lugar do computador (CPU) que se chama **acumulador**. **Acumulador?!?** O que é afinal de contas um **acumulador**? O acumulador é um lugar do computador capaz de guardar números. Este acumulador funciona como uma variável: toda vez que um valor é colocado nele, o último valor é perdido. Então, como o resultado final é colocado na variável `potx`? Depois que a função encontrou um **return**, o fluxo do programa volta para o comando `potx = potencia (x, a);`. Neste momento, o resultado da função está no **acumulador**, pois o código da função `potencia` já foi executado e consequentemente o comando **return** também já foi executado (aliás, este é o último comando executado da função). Aí, como temos indicado que a variável `potx` recebe `potencia (x, a)` no comando `potx = potencia (x, a);`, então o valor que está no **acumulador** é colocado em `potx`.

Da mesma forma, o uso `poty = potencia (y, b);` faz com que `expoente ← b` e `base ← y` e executa o código de `potencia`; no final da execução, a variável `pot` contém o valor de y^b; o comando **return** `pot;` coloca o valor que está na variável `pot` da função `potencia` no **acumulador**. Depois que a função encontrou um **return**, o fluxo do programa volta para o comando `poty = potencia (y, b);`. Neste momento, o resultado da função está no **acumulador**, pois o código da função `potencia` já foi executado e consequentemente o comando **return** também já foi executado (aliás, este é o último comando executado da função). Aí, como temos indicado que a variável `poty` recebe `potencia (y, b)` no comando `poty = potencia (y, b);`, então o valor que está no **acumulador** é colocado em `poty`.

Mais uma vez, o uso `potxy = potencia (x-y, a+b);` faz com que `expoente ← a+b` e `base ← x-y` e executa o código de `potencia`; no final da execução, a variável `pot` contém o valor de $(x-y)^{a+b}$; o comando **return** `pot;` coloca o valor que está na variável `pot` da função `potencia` no **acumulador**. Depois que a função encontrou um **return**, o fluxo do programa volta para o comando `potxy = potencia (x-y, a+b);`. Neste momento, o resultado da função está no **acumulador**, pois o código da função `potencia` já foi executado e consequentemente o comando **return** também já foi executado (aliás, este é o último comando executado da função). Aí, como

temos indicado que a variável `potxy` recebe `potencia (x-y, a+b)` no comando `potxy = potencia (x-y, a+b);`, então o valor que está no **acumulador** é colocado em `potxy`.

Você deve estar se perguntando: como coloco tudo isto em um programa em C? Veja o código em C abaixo:

```
1   # include <stdio.h>
2
3
4
5   float potencia (float base, int expoente) {
6     float pot;
7     int cont;
8     cont = 0; pot = 1;
9     while (cont < expoente) {
10      pot = pot * base;
11      cont = cont + 1;
12    }
13    /* no final pot contém base elevado a expoente */
14    return pot;
15  }
16
17  int main () {
18    float x, y, potx, poty, potxy, soma;
19    int a, b;
20
21    printf ("Entre com dois reais x e y: ");
22    scanf ("%f %f", &x, &y);
23
24    printf ("Entre com dois inteiros a>0 e b>0: ");
25    scanf ("%d %d", &a, &b);
26
27
28    potx = potencia (x, a);
29
30
31    poty = potencia (y, b);
32
33
34    potxy = potencia (x-y, a+b);
35
36    soma = potx + poty + potxy;
37    printf ("Resultado = %f\n", soma);
38
39    return 0;
40  }
```

Note que antes do nome da função `potencia` foi colocado um **float**. Este **float** indica que a função "vai jogar para fora" (devolver) um valor do tipo **float** via **return**. De fato, observe que no comando `return pot;`, a variável `pot` guarda um valor real.

A idéia de função no exercício anterior é fazer um subprograma de nome `potencia` que seja utilizado três vezes. Este subprograma deve ter entrada e saída. Por exemplo, para calcular x^a, a entrada do subprograma deve ser o real x e o inteiro a e a saída deve ser o real x^a.

Em resumo, o uso de funções facilita a construção de programas pois possibilita a reutilização de partes de código.

15.3 Exercícios Recomendados

1. Considerando a função fatorial, que parâmetros ela deve receber (incluindo os tipos)? E qual o tipo do valor de saída (real ou inteiro?).

2. Considere uma função `seno` que calcula o valor do seno de um ângulo x com precisão $epsilon$. Que parâmetros ela deve receber (incluindo os tipos)? E qual o tipo do valor de saída dessa função(real ou inteiro?).

3. Escreva um programa que leia dois inteiros `m` e `n`, com `m`$\geq$ `n`, e calcula

$$\texttt{C(m,n)} = \frac{\texttt{m!}}{\texttt{n!(m-n)!}}$$

16 Definição e Uso de Funções em Programas

Ronaldo F. Hashimoto e Carlos H. Morimoto

Nessa aula falaremos mais sobre funções. Veremos como declarar funções usando protótipos, como definir o corpo da função e como utilizar funções em seus programas (chamada de função e passagem de parâmetros).

Ao final dessa aula você deverá saber:

- Escrever o protótipo de uma função.
- Escrever o corpo de uma função.
- Utilizar funções em seus programas.
- Simular a execução de programas com funções.

16.1 Função Principal

Um programa na linguagem C pode ser organizado na forma de funções, onde cada função é responsável pelo cálculo/processamento de uma parte do programa. Por exemplo, no exercício da aula passada, a função `potencia` era responsável pelo cálculo das potenciações envolvidas no programa que era a solução do exercício.

Em particular, o `main` é uma função! Todo programa em C precisa de uma função chamada `main` (função principal), que, dentre todas as funções que estão em seu programa, ela é a primeira a ser executada.

Para efeitos deste curso, quando um exercício de Introdução à Computação pedir para você **fazer um programa**, isto significa que estamos pedindo para você escrever a **função principal**, ou seja, a função `main`, como você tem feito até agora. Só que agora, sua função `main` vai fazer uso de outras funções definidas por você ou que fazem parte de alguma biblioteca de funções.

16.1.1 Entrada e Saída da Função Principal

Já que estamos falando da função `main`, vamos relembrar um pouco sobre entrada e saída da função principal.

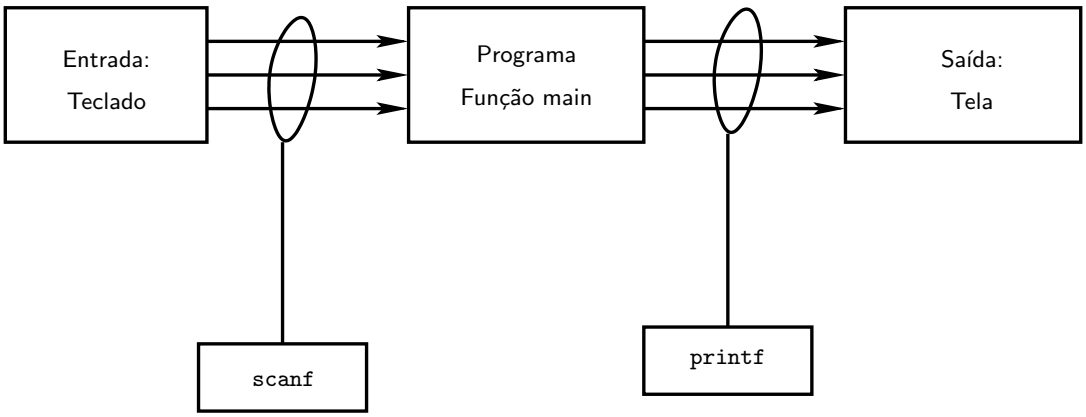


Figura 13: Entrada e Saída do Programa Principal.

A entrada de dados do programa principal (ou seja, do "**int** `main ()`") é feita pelo teclado e para isso é usado o comando de leitura `scanf` (veja Fig. 13).

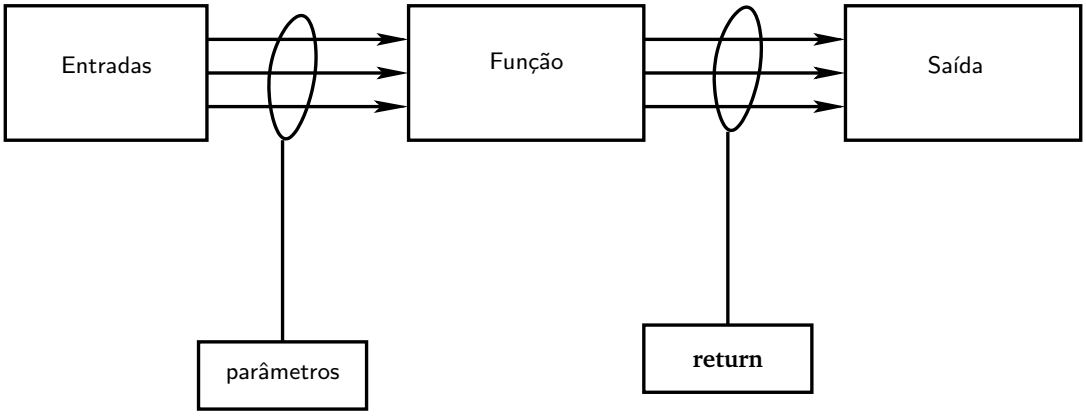


Figura 14: Entrada e Saída de Funções.

A saída de dados do programa principal é feita por impressões na tela e para isso é usado o comando de impressão `printf` (veja Fig. 13).

Ora, por que estamos falando tudo isso? Porque para outras funções que não seja a função principal, a entrada e saída dos dados normalmente não é feita pelo teclado e nem pela tela. Para entrada de dados, usamos o que chamamos de **parâmetros** e para saída, por enquanto, usaremos o comando **return** (veja Fig. 14).

16.2 As Outras Funções

As "outras funções" são as funções que você usa no seu programa em C com exceção da função principal. Para estas funções, a entrada de dados normalmente é feita por parâmetros. No exemplo da aula anterior, os parâmetros são a base e o expoente da potenciação que se deseja calcular. Para fins deste curso, a saída das "outras funções" é um valor (e somente um único valor) que pode ser um **int**, **float** ou **char**. No exemplo da aula anterior, a saída é um valor do tipo **float** (feita via **return**) e corresponde ao resultado da potenciação que se deseja calcular.

Isto significa que normalmente (note que está escrito **normalmente**) não devemos colocar `scanf` e nem `printf` em funções que não sejam a função principal.

Este conceito de **parâmetros** e **return** para as "outras funções" é muito importante. Se você não entendeu, por favor, releia com atenção a primeira aula de funções. Tem muito aluno que coloca `scanf` e `printf` nas "outras funções" significando que este conceito não foi bem entendido.

16.3 Funções Definidas por Você

Para definir uma função, você deve ter em claro quais são as entradas (parâmetros) e qual é a saída (que valor e o tipo do mesmo que a função deve devolver via **return**). Para o exemplo da aula anterior, a função `potencia` tem dois parâmetros: `base` e `expoente` e a saída é um valor **float** resultado da potenciação `base` elevado a `expoente` (veja Fig. 15).

16.3.1 Cabeçalho ou Protótipo de uma Função

O cabeçalho ou protótipo de uma função contém o tipo do valor devolvido (**int**, **char** ou **float**), o nome da função e uma lista de parâmetros, entre parênteses, seguido de um ponto e vírgula dispostos da seguinte forma:

```
<tipo_do_retorno> <nome_da_função> (<lista_de_parâmetros>);
```

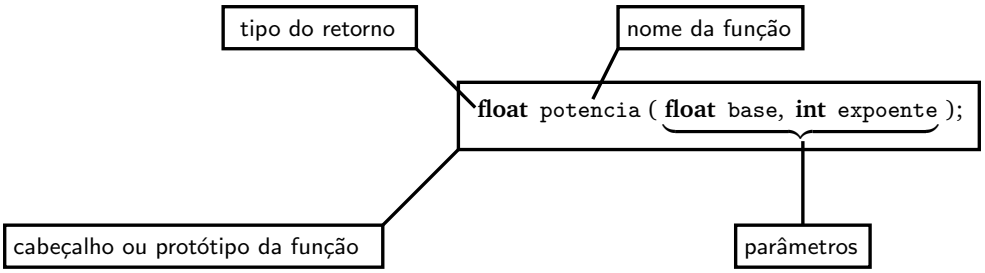


Figura 15: Entrada e Saída da Função `potencia`.

onde

- <tipo_do_retorno>: define o tipo do valor devolvido pela função (saída) via **return**;
- <nome_da_função>: nome da função;
- <lista_de_parâmetros>: parâmetros de entrada da função.

Mais uma vez, note que a <lista_de_parâmetros> está entre parênteses e o cabeçalho termina com um ponto e vírgula.

A forma de escrever a <lista_de_parâmetros> é a mesma forma de declarar variáveis, ou seja, na lista de parâmetros as variáveis são declaradas como <tipo_da_variável> <nome_da_variável> separadas por vírgulas.

Por exemplo, considere uma função `potencia` da aula passada que calcula `base` elevado a `expoente`, onde é um `base` número real e `expoente` é um número inteiro. Como o resultado de `base` elevado a `expoente` é um número real (uma vez que `base` é um número real), então a função deve ser do tipo **float**. Dessa forma, o seu protótipo seria:

```
float potencia (float base, int expoente);
```

16.3.2 Como Definir uma Função: Corpo de uma Função

Uma função é **definida** com o seu cabeçalho (sem o ponto e vírgula) seguido por um bloco de comandos definidos entre chaves. Chamamos este bloco de comandos dentro da função como *corpo da função*. Por exemplo, no caso da função `potencia`, como vimos na aula passada, sua definição poderia ser:

```
1   potencia (float base, int expoente) {
2     float pot;
3     int cont;
4     cont = 0; pot = 1;
5     while (cont < expoente) {
6       pot = pot * base;
7       cont = cont + 1;
8     }
9     /* no final pot contém base elevado a expoente */
10    return pot;
11  }
```

Dentro de cada função, você deve primeiramente declarar as variáveis que serão utilizadas **dentro** da função (ou seja entre as chaves que definem o corpo da função). Dentro da função, os parâmetros de entrada equivalem a variáveis, porém, o valor inicial dessas "variáveis" são definidas na hora de usar (chamar) a função no programa principal, por exemplo.

Os comandos que definem a função são definidos dentro do bloco, ao final, o resultado da função é devolvido ao local da chamada através do comando **return**.

16.4 Exercício

Vamos fazer um exercício para fixar estas idéias. Este exercício está dividido em três partes:

(a) Escreva uma função **int** `fat` (**int** `n`) que recebe como parâmetro um inteiro `n` e calcula o fatorial de `n`.

Solução:

```
1   int fat (int n) {
2     int cont = 1, fatorial = 1;
3
4     while (cont <= n) {
5       fatorial = fatorial * cont;
6       cont = cont + 1;
7     }
8
9     return fatorial;
10  }
```

(b) Utilizando a função do item anterior, faça uma função **int** `comb` (**int** `m`, **int** `n`) que recebe dois inteiros `m` e `n`, com `m`$\geq$ `n`, e calcula

$$\texttt{C(m,n)} = \frac{\texttt{m!}}{\texttt{n!(m-n)!}}$$

Solução:

```
1   int comb (int m, int n) {
2     int resultado;
3
4     resultado = fat (m) / (fat (n)*fat(m-n));
5
6     return resultado;
7   }
```

(c) Utilizando a função do item anterior, faça um programa que leia (do teclado) um inteiro `n > 0` e imprime (na tela) os coeficientes da expansão de $(a+b)^{\texttt{n}}$.

Exemplo: `n = 2` $\Rightarrow$ `1, 2, 1`
`n = 4` $\Rightarrow$ `1, 4, 6, 4, 1`

O nosso programa deve fazer:

para `n = 2`, imprimir: `C(2,0), C(2,1), C(2,2)`
para `n = 4`, imprimir: `C(4,0), C(4,1), C(4,2), C(4,3), C(4,4)`
para `n = x`, imprimir: `C(x,0), C(x,1),.....,C(x,x)`

Solução:

```
1   # include <stdio.h>
2
3   int main () {
4
5     int n, coef, cont;
6
7     printf ("Entre com n > 0: ");
8     scanf ("%d", &n);
9     cont  = 0;
10    while (cont <= n) {
11      coef = comb (n, cont);
12      printf ("%d\n", coef);
13      cont = cont + 1;
14    }
15
16    return 0;
17  }
```

Note que a leitura pelo teclado (`scanf`) e a impressão na tela (`printf`) estão somente na função principal. Nas funções `fat` e `comb` a entrada de dados é feita via parâmetros e a saída via **return**.

16.5 Estrutura de um Programa

A estrutura de um programa com funções deve seguir a seguinte forma:

```
/* includes */

/* protótipos das funções */

/* definição das funções */

/* função principal */
```

Nosso programa do exercício anterior com funções seria:

```
1   /* includes */
2
3   # include <stdio.h>
4
5   /* protótipos das funções */
6
7   int fat (int n);
8   int comb (int m, int n);
9
10  /* definição das funções */
11
12  int fat (int n) {
13    int cont = 1, fatorial = 1;
14
15    while (cont <= n) {
16      fatorial = fatorial * cont;
17      cont = cont + 1;
18    }
19
20    return fatorial;
21  }
22
23  int comb (int m, int n) {
24    int resultado;
25
26    resultado = fat (m) / (fat (n)*fat(m-n));
27
28    return resultado;
29  }
30
31  /* função principal */
32
33  int main () {
34
35    int n, coef, cont;
36
37    printf ("Entre com n > 0: ");
38    scanf ("%d", &n);
39    cont  = 0;
40    while (cont <= n) {
41      coef = comb (n, cont);
42      printf ("%d\n", coef);
43      cont = cont + 1;
44    }
45
46    return 0;
47  }
```

Na realidade, a declaração explicita dos protótipos da funções (linhas 7 e 8 do programa) não é necessária, mas é uma boa prática de programação pois permite que você defina as funções em qualquer ordem.

Caso os protótipos não sejam declarados, as funções precisam ser definidas antes de serem utilizadas por outras funções. Por exemplo, no nosso exemplo, como a função `comb` usa (chama) a função `fat`, então a função `fat` deve ser definida antes da função `comb`.

16.6 Simulação de Funções

Simule a saída do seguinte programa:

```
1   # include <stdio.h>
2
3   int g (int x, int y) {
4     int z;
5     z = x * x + y * y;
6     return z;
7   }
8
9   int h (int x, int a) {
10    int d;
11    d = g (a, x) + 3;
12    return d;
13  }
14
15  int main () {
16    int y, z, x;
17    z = 2;
18    y = 3;
19    x = h (y, z);
20    printf ("x = %d\n", x);
21
22    return 0;
23  }
```

O programa sempre começa a ser executado na função principal.

	main		
	y	z	x
Início →	?	?	?
Linhas 17 e 18 →	3	2	?

Na Linha 19 é chamada a função `h`. Note a passagem de parâmetros:

- `y` da função `main` → (para) `x` da função `h` e
- `z` da função `main` → (para) `a` da função `h`

	main		
	y	z	x
Início →	?	?	?
Linhas 17 e 18 →	3	2	?

	h		
	x	a	d
Início →	?	?	?
Linha 9 →	3	2	?

Note que a variável `x` da `main` não tem nada a ver com a variável `x` da função `h`. Cada função tem sua variável, ou seja, mesmo sendo de mesmo nome, as variáveis são diferentes, pois estão declaradas em funções diferentes.

Na Linha 11 é chamada a função `g`. Note a passagem de parâmetros:

- `a` da função `h` → (para) `x` da função `g` e
- `x` da função `h` → (para) `y` da função `g`

	h		
	x	a	d
Início →	?	?	?
Linha 9 →	3	2	?

	g		
	x	y	z
Início →	?	?	?
Linha 9 →	2	3	?

Na Linha 5, calcula `z` da função `g`:

	g		
	x	y	z
Início →	?	?	?
Linha 9 →	2	3	13

Na Linha 6, temos um **return**. Na aula passada vimos que o **return**:

- guarda o resultado ou o conteúdo de uma variável no ACUM.
- finaliza a função. O fluxo do programa volta para onde a função foi chamada.

Neste caso, coloca o valor de `z` da função `g` no ACUM e volta para a Linha 11.

	ACUM
Início →	?
Linha 11 →	13

Na Linha 11, calcula `d` da função `h`:

	h		
	x	a	d
Início →	?	?	?
Linha 9 →	3	2	16

Na Linha 12, coloca o valor de `d` da função `h` no ACUM e volta para a Linha 19.

	ACUM
Início →	?
Linha 11 →	13
Linha 12 →	16

Na Linha 19, coloca o conteúdo do ACUM na variável `x` da função `main`:

	main		
	y	z	x
Início →	?	?	?
Linhas 17 e 18 →	3	2	?
Linhas 19 →	3	2	16

No final imprime na tela o contéudo da variável `x` da função `main` na tela:

Tela
x = 16

16.7 Funções que fazem parte de alguma Biblioteca

Existe um conjunto de funções pré-definidas na linguagem C que você pode usar. Particularmente, você pode estar interessado em funções matemáticas tais como funções trigonométricas, hiperbólicas, logarítmicas, exponenciais, etc... A seguir, listamos algumas destas funções[4]:

```
/*****************************************************
Seção 1 -- Funções trigonométricas
*****************************************************/

double sin (double x);
double cos (double x);
double tan (double x);

/*****************************************************
Seção 2 -- Exponenciais e logaritmos
*****************************************************/

/* Uso típico: y = exp (x);                          */
/* Devolve e^x, ou seja, o número e elevado à        */
/* potência x.                                       */

double exp (double x);


/* Uso típico: y = log (x);                          */
/* Devolve o logaritmo de x na base e. Não use com   */
/* x negativo.                                       */

double log (double x);


/* Uso típico: y = log10 (x);                        */
/* Devolve o logaritmo de x na base 10. Não use com  */
/* x negativo.                                       */

double log10 (double x);

/*****************************************************
Seção 3 -- Raiz e potência
*****************************************************/

/* Uso típico: y = sqrt (x);                         */
/* Devolve a raiz quadrada de x. Não use com x < 0.  */

double sqrt (double x);


/* Uso típico: p = pow (x, y);                       */
/* Devolve x^y, ou seja, x elevado à potência y.     */
/* Não use com x = 0.0 e y < 0.0. Não use com x < 0.0 */
/* e y não-inteiro.                                  */
/* Caso especial: pow (0.0, 0.0) == 1.0.             */
/* Que acontece se x^y não couber em double? Veja man */
/* pages.                                            */

double pow (double x, double y);
```

[4]retiradas da página <http://www.ime.usp.br/~pf/algoritmos/apend/math.h.html>

```
/*****************************************************
Seção 4 - Valor Absoluto
*****************************************************/

/* Uso típico: i = fabs (x). A função devolve o valor  */
/* absoluto de x.                                      */


double fabs (double x);


/*****************************************************
Seção 5 -- Arredondamentos
*****************************************************/

/* Uso típico: i = floor (x). A função devolve o maior */
/* inteiro que seja menor que ou igual a x, isto é,    */
/* o único inteiro i que satisfaz i <= x < i+1.        */

double floor (double x);


/* Uso típico: j = ceil (x). A função devolve o menor  */
/* inteiro que seja maior que ou igual a x, isto é,    */
/* o único inteiro j que satisfaz j-1 < x <= j.        */

double ceil (double x);
```

Para podermos utilizar essas funções, devemos incluir o arquivo de cabeçalhos # **include** <math.h> no início do seu programa em C. Além disso, se você estiver usando o compilador gcc, você precisa compilar o seu programa com a opção `-lm`:

```
gcc -Wall -ansi -pedantic -O2 <nome_do_programa>.c -o <nome_do_programa> -lm
```

16.8 Exercícios Recomendados

1. Um número a é dito **permutação** de um número b se os dígitos de a formam uma permutação dos dígitos de b. Exemplo: 5412434 é uma permutação de 4321445, mas não é uma permutação de 4312455. Obs.: Considere que o dígito 0 (zero) não aparece nos números.
   - Faça uma função `contadígitos` que dados um inteiro n e um inteiro d, $0 < d \leq 9$, devolve quantas vezes o dígito d aparece em n.
   - Usando a função do item anterior, faça um programa que lê dois números a e b e responde se a é permutação de b.
2. Nesse exercício você deve utilizar a função `sqrt(x)` do `math.h`
   - Escreva uma função real que recebe dois pontos no plano através de suas coordenadas cartesianas e devolve a distância entre os pontos.
   - Faça um programa que leia um ponto origem (x_0, y_0) e uma sequência de n pontos e determina o ponto mais próximo da origem, usando a função do item anterior.
3. Programa que verifica se um número pode ser escrito como soma de 2 números primos.
   - Faça uma função de nome `primo` que recebe um inteiro `n > 1` e verifica se `n` é primo ou não. Esta função devolve `0` (zero) se `n` não é primo; e devolve `1` (um) se `n` é primo.

- Faça um programa que leia um inteiro `n > 2` e verifica se `n` pode ser escrito como `p + q`, com `p > 1`, `q > 1`, `p` e `q` primos.
  Por exemplo, n = 5 ⇒ SIM, pois 5 = 2 + 3
  n = 11 ⇒ NÃO.
  Note que seu programa deve testar se existe um `p` primo tal que `p>0`, `p<n` e `n-p` é primo.

17 Funções e Ponteiros

Ronaldo F. Hashimoto e Carlos H. Morimoto

Nessa aula introduzimos o mecanismo utilizado pelo C para permitir que funções devolvam mais de um valor, que é realizando através de variáveis do tipo ponteiros (ou apontadores). Para isso, vamos primeiramente descrever como a informação é armazenada na memória do computador, para definir o que é um ponteiro e sua utilidade.

Ao final dessa aula você deverá ser capaz de:

- Descrever como a informação é armazenada na memória do computador, e a diferença entre conteúdo e endereço de uma posição de memória.
- Definir o que é um ponteiro (ou apontador), e para que ele é utilizado.
- Declarar ponteiros.
- Utilizar ponteiros como parâmetros de funções e em programas.

17.1 Memória

Podemos dizer que a memória (RAM) de um computador pode ser constituída por **gavetas**. Para fins didáticos, considere um computador imaginário que contenha na memória 100 (cem) gavetas (veja na Figura 16 uma idéia do que pode ser uma memória de um computador).

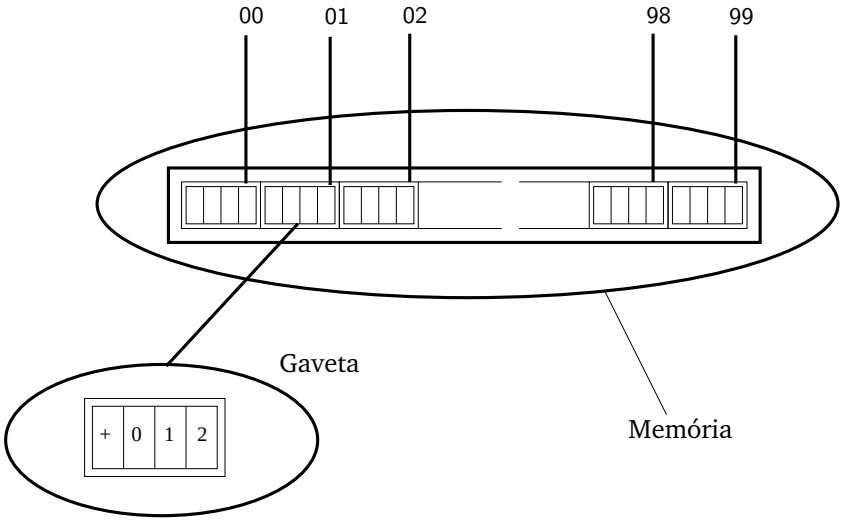


Figura 16: Memória de um Computador

Cada gaveta é dividida em regiões. Por questões didáticas, vamos considerar que cada gaveta tem 4 (quatro) regiões. Na região mais à esquerda pode-se colocar um sinal '+' ou '–'. Nas demais quatro regiões, dígitos de `0` a `9`. Assim, em uma gaveta, podemos guardar números de `-999` a `+999`.

Em computadores reais, cada gaveta tem 8 (oito) regiões (estas regiões são chamadas de *bits*). O bit mais à esquerda é conhecido como **bit de sinal**: 0 (zero) para números positivos e 1 (um) para números negativos. Estes 8 bits compõem o que chamamos de *byte*. Assim uma gaveta tem tamanho de 1 (um) byte. Dessa forma, uma gaveta guarda um número (que em computadores reais é uma sequência de 8 zeros e uns).

Agora, o mais importante: para cada gaveta na memória é associada uma identificação numérica (no exemplo da Fig. 16, esta identificação corresponde à numeração das gavetas de 00 a 99, mas em computadores reais,

o último número pode ser muito maior que 99). Estes números identificam as gavetas e são chamados de **endereços**. Assim, cada gaveta tem um endereço associado indicando a posição onde ela se encontra na memória.

17.2 Declaração de Variáveis

Quando se declara uma variável, é necessário reservar espaço na memória que seja suficiente para armazená-la, o que pode corresponder a várias gavetas dependendo do tipo da variável. Para variáveis inteiras são reservadas 4 gavetas (bytes). Para **float** e **double** são reservadas 4 e 8 bytes, respectivamente.

Dessa forma, as seguintes declarações

```
int n;
float y;
double x;
```

reservam 4, 4 e 8 bytes para a variáveis `n`, `y` e `x`, respectivamente.

Agora, vamos considerar a variável inteira `n`. Esta variável tem então 4 gavetas com endereços, digamos, `20`, `21`, `22` e `23` (veja a Fig. 17).

Considerando a região mais à esquerda como um sinal '+' ou '–' e nas demais 15 (quinze) regiões, dígitos de `0` a `9`, essa variável inteira `n` no computador imaginário pode armazenar qualquer número inteiro que está no intervalo de `-999.999.999.999.999` a `+999.999.999.999.999`. Em um computador real, a variável `n` pode armazenar qualquer número inteiro de `-2.147.483.648` a `+2.147.483.647`.

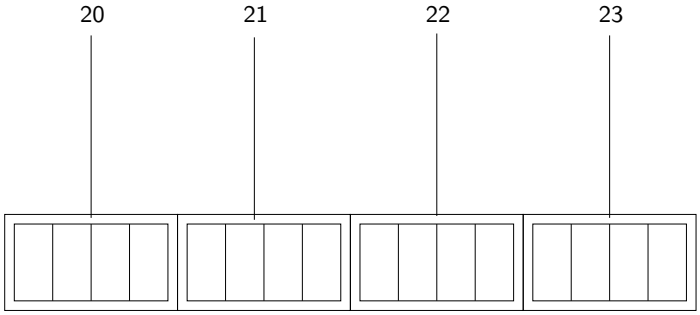


Figura 17: Quatro gavetas para a variável `n`

Agora, outra coisa importantíssima: toda variável é associada a um endereço. Para a variável `n`, apesar de ter 4 gavetas, ela é associada somente a um endereço. Vamos supor que seja o menor deles, ou seja, neste caso, o endereço 20.

Bem, por que estamos falando dessas coisas de **endereço de variáveis** se a matéria desta aula é **ponteiros**? É porque um **ponteiro** (ou **apontador**) é um tipo de variável que consegue guardar **endereço de variável**. Assim, é possível guardar o endereço da variável `n` em uma variável do tipo **ponteiro**.

Vamos aqui fazer uma suposição de que temos uma variável **ponteiro** `apt`. Legal, esta variável pode guardar endereços de variáveis. Então, como fazer com que a variável `apt` guarde o endereço de `n`? Em C, o seguinte comando

```
apt = &n;
```

faz com que a variável `apt` guarde o endereço de `n`. O operador `&` antes do nome da variável significa "endereço de". Assim, o comando de atribuição acima coloca em `apt` o "endereço de" de `n`.

Se uma variável do tipo **ponteiro** `apt` guarda o endereço da variável `n`, então podemos dizer que a variável `apt` **aponta** para a variável `n`, uma vez que `apt` guarda o local onde `n` está na memória. Você pode estar se perguntando para que serve tudo isso. Observe que ter uma variável que guarda o local de outra variável na memória pode dar um grande poder de programação. Isso nós vamos ver mais adiante ainda nesta aula.

17.3 Declaração de Variável Tipo Ponteiro

Uma outra coisa importante aqui é saber que ponteiros de variáveis inteiras são diferentes para ponteiros para variáveis do tipo **float** e **char**.

Para declarar uma variável ponteiro para **int**:

```
int * ap1;
```

Para declarar uma variável ponteiro para **float**:

```
float * ap2;
```

Para declarar uma variável ponteiro para **char**:

```
char * ap3;
```

As variáveis `ap1`, `ap2` e `ap3` são **ponteiros** (guardam endereços de memória) para **int**, **float** e **char**, respectivamente.

17.4 Uso de Variáveis Tipo Ponteiros

Para usar e manipular variáveis do tipo ponteiros, é necessário usar dois operadores:

- `&` : **endereço de**
- `*` : **vai para**

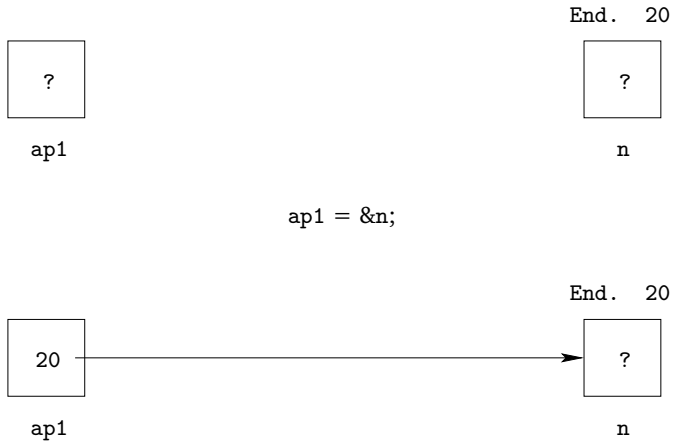


Figura 18: Operador &(“endereço de”).

17.4.1 Uso do Operador “endereço de”

Para uma variável ponteiro receber o endereço de uma outra variável é necessário usar o operador & (“endereço de”). Observe os seguintes exemplos:

```
ap1 = &n;    /* ap1 recebe o endereço da variável n */
ap2 = &y;    /* ap2 recebe o endereço da variável y */
ap3 = &x;    /* ap3 recebe o endereço da variável x */
```

Observe a Fig. 18. A execução do comando `ap1 = &n;` faz com que a variável `ap1` receba o endereço da variável `n`. Uma vez que a a variável `ap1` guarda o endereço da variável `n` é como se `ap1` estivesse apontando para a variável `n`. Por isso que variáveis que guardam endereços são chamadas de variáveis do tipo ponteiro.

17.4.2 Uso do Operador “vai para”

Uma vez que o ponteiro está apontando para uma variável (ou seja, guardando seu endereço) é possível ter acesso a essa variável usando a variável ponteiro usando o operador $*$ (“vai para”):

```
* ap1 = 10;   /* vai para a gaveta que o ap1 está apontando e guarde  10 nesta gaveta */
* ap2 = -3.0; /* vai para a gaveta que o ap2 está apontando e guarde -3.0 nesta gaveta */
* ap3 = -2.0; /* vai para a gaveta que o ap3 está apontando e guarde -2.0 nesta gaveta */
```

Observe a Fig. 19. A execução do comando `*ap1 = 10;` diz o seguinte: vai para a gaveta que o `ap1` está apontando (ou seja, a variável `n`) e guarde `10` nesta gaveta, ou seja, guarde `10` na variável `n`.

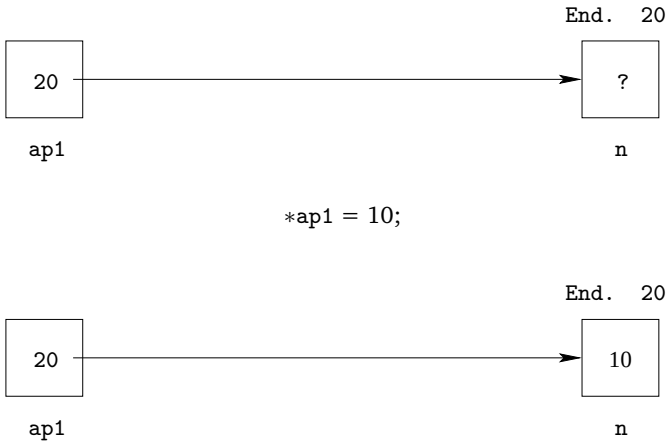


Figura 19: Operador * (“vai para”).

Uma observação importantíssima: o uso do $*$ é diferente quando a gente declara de variável do tipo ponteiro e quando a gente usa como operador “vai para”. Observe com cuidado o seguinte código:

```
int main () {
  int n, m;
  float y;
  char x;

  int * ap1;
  float * ap2;
  char * ap3;

  n = 2; m = 3;
  y = 5.0; x = 's';

  ap1 = &n;
  ap2 = &y;
  ap3 = &x;

  * ap1 = m;
  * ap2 = -5.0;
  * ap3 = 'd';

  printf ("n = %d y = %f x = %c\n", n, y, x);
  return 0;
}
```

Declaração das variáveis ponteiros. Note o uso do asterisco para declarar ponteiros.

Uso de Ponteiros: & significa "endereço de". Note que aqui não tem o asterisco antes da variável ponteiro.

Uso de Ponteiros: Note que aqui usa-se o asterisco antes do ponteiro. Este asterisco significa "vai para"a gaveta que o ponteiro está indicando (apontando).

O que o último `printf` vai imprimir na tela? Se você não entendeu o código acima, digite, compile e execute e veja sua saída.

17.5 Para que serve Variáveis Ponteiros?

Você ainda deve se estar perguntando: para que tudo isso?!? Observe atentamente o seguinte código para fins didáticos:

```
1   # include <stdio.h>
2
3
4   int f (int x, int *y) {
5     *y = x + 3;
6     return *y + 4;
7   }
8
9   int main () {
10    int a, b, c;
11
12    a = 5; b = 3;
13
14    c = f (a, &b);
15
16    printf ("a = %d, b = %d, c = %d\n", a, b, c);
17
18    return 0;
19  }
```

O programa sempre começa a ser executado na função principal.

	main		
	a	b	c
Início →	?	?	?
Linha 12 →	5	3	?

Na Linha 14 é chamada a função `f`. Note a passagem de parâmetros:

- `a` da função `main` → (para) `x` da função `f` e
- `&b` da função `main` → (para) `y` da função `f`

Note que no segundo parâmetro, a variável `y` da função `f` é um ponteiro para **int**. Note que a variável `y` recebe o endereço da variável `b` da função `main`. Dessa forma, a variável `y` da função `f` aponta para a variável `b` da função `main`.

	main		
	a	b	c
Início →	?	?	?
Linha 12 →	5	3	?

	f	
	x	y
Início →	?	?
Linha 4 →	5	

Na Linha 5, estamos usando o ponteiro `y` da função `f` com o operador "vai para". Ou seja, "vai para" a gaveta em que o `y` está apontando e guarde o resultado da expressão `x+3` neste lugar. Como `y` está apontando para a variável `b` da função `main`, então o resultado da expressão `x+3` será guardado na variável `b` da função `main`. Depois da execução da Linha 5, temos então:

	main		
	a	b	c
Início →	?	?	?
Linha 12 →	5	~~3~~ 8	?

	f	
	x	y
Início →	?	?
Linha 4 →	5	

Na Linha 6 temos um **return**. Neste **return**, temos que calcular o valor da expressão `*y+4`. O `*y+4` significa: "vai para" a gaveta em que o `y` está apontando, pegue o valor inteiro que está lá e some com 4 (quatro). Como `y` está apontando para `b` da `main`, então a conta que deve ser feita é: pegue o valor de `b` e some com `4` (quatro). Como `b` vale `8`, então o resultado da expressão `*y+4` é `12` (doze). Este resultado é colocado no ACUM.

	ACUM
Início →	?
Linha 6 →	12

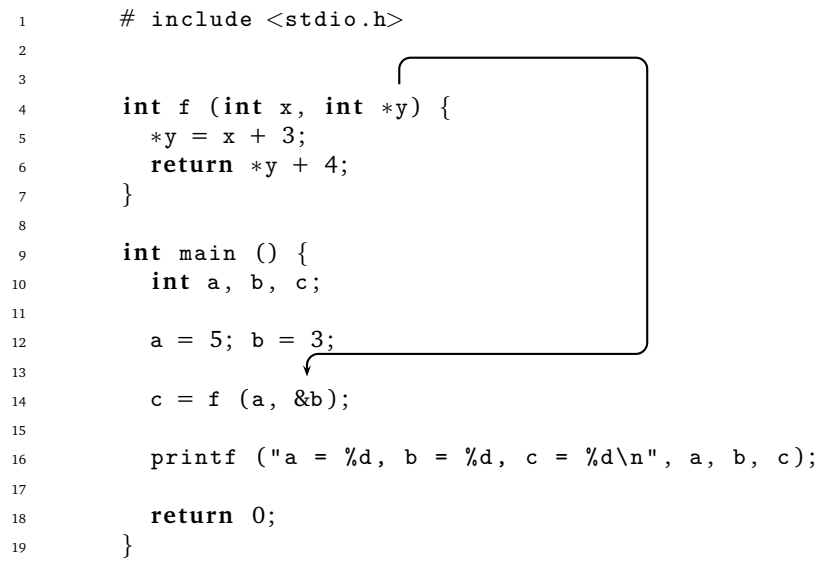
e o fluxo do programa volta para onde a função `f` foi chamada, ou seja, na Linha 14. Neste linha, como temos um comando de atribuição para a variável `c`, esta variável recebe o valor que está em ACUM, ou seja, recebe o valor 12.

	main		
	a	b	c
Início →	?	?	?
Linha 12 →	5	~~3~~ 8	?
Linha 14 →	5	8	12

No final, o programa imprime na tela o contéudo das variáveis `a`, `b` e `c` da função `main`:

Tela
a = 5, b = 8, c = 12

Note que na verdade então, o parâmetro `y` da função `f` (que é um ponteiro pois está declarado com um asterisco) aponta para a variável `b` da função `main`, uma vez que na chamada da função `f`, a variável `b` é o segundo parâmetro da função `f`. Dessa forma, podemos dizer que `y` da função `f` aponta para `b` da função `main`:



```
1   # include <stdio.h>
2
3
4   int f (int x, int *y) {
5     *y = x + 3;
6     return *y + 4;
7   }
8
9   int main () {
10    int a, b, c;
11
12    a = 5; b = 3;
13
14    c = f (a, &b);
15
16    printf ("a = %d, b = %d, c = %d\n", a, b, c);
17
18    return 0;
19  }
```

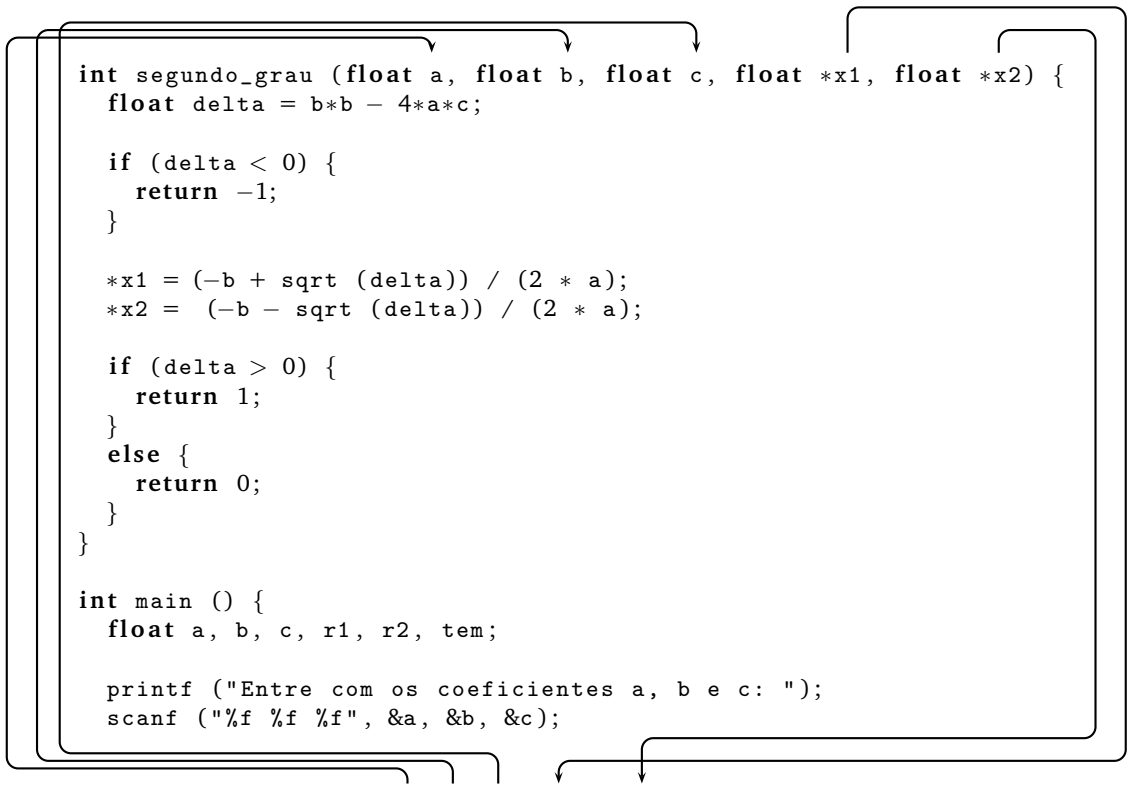
E todas as modificações que fizermos em `*y`, estamos na realidade fazendo na variável `b`.

Legal! E para que serve isso? Os ponteiros como **parâmetros de função** (como o parâmetro `y` da função `f`) servem para **modificar** as variáveis que estão **fora** da função (como a variável `b` da função `main`). E para que serve isso? Considere o seguinte problema:

17.6 Problema

Faça um função que recebe como parâmetros de entrada três reais `a`, (`a` $\neq 0$), `b` e `c` e resolve a equação de 2o. grau $\boxed{\texttt{a}x^2+\texttt{b}x+\texttt{c}=0}$ devolvendo as raízes em dois ponteiros `*x1` e `*x2`. Esta função ainda deve devolver via **return** o valor -1 se a equação não tem raízes reais, 0 se tem somente uma raiz real e 1 se a equação tem duas raízes reais distintas.

Note que a função pedida deve devolver **dois** valores que são as raízes da equação de 2o. grau $\boxed{\texttt{a}x^2+\texttt{b}x+\texttt{c}=0}$. Mas nós aprendemos que uma função só consegue devolver **um único** valor via **return**. Como então a função vai devolver dois valores? Resposta: usando ponteiros!



```c
#include <stdio.h>
#include <math.h>

int segundo_grau (float a, float b, float c, float *x1, float *x2) {
  float delta = b*b - 4*a*c;

  if (delta < 0) {
    return -1;
  }

  *x1 = (-b + sqrt (delta)) / (2 * a);
  *x2 =  (-b - sqrt (delta)) / (2 * a);

  if (delta > 0) {
    return 1;
  }
  else {
    return 0;
  }
}

int main () {
  float a, b, c, r1, r2, tem;

  printf ("Entre com os coeficientes a, b e c: ");
  scanf ("%f %f %f", &a, &b, &c);

  tem = segundo_grau (a, b, c, &r1, &r2);


  if (tem < 0) {
    printf ("Eq. sem raizes reais\n");
  }

  if (tem == 0) {
    printf ("Eq. tem somente uma raiz\n");
    printf ("raiz = %f\n", r1);
  }

  if (tem > 0) {
    printf ("Eq. tem duas raizes distintas\n");
    printf ("Raízes %f e %f\n", r1, r2);
  }

  return 0;
}
```

Note que neste problema, os parâmetros `x1` e `x2` são ponteiros que modificam as variáveis `r1` e `r2` da funcão `main`. Neste caso então, podemos colocar as soluções da equação de 2o. grau em `*x1` e `*x2` (da forma como foi feita na solução do problema) que estamos na realidade colocando as soluções nas variáveis `r1` e `r2` da funcão `main`. Faça uma simulação da solução do problema e verifique que as soluções de fato ficam armazenadas nas variáveis `r1` e `r2` da funcão `main`.

17.7 Função do Tipo void

Funções do tipo **void** correspondem a funções que não retornam um valor via **return**. Veja o seguinte exemplo didático:

Você ainda deve se estar perguntando: para que tudo isso?!? Observe atentamente o seguinte código para fins didáticos:

```
1   # include <stdio.h>
2
3
4   void g (int x, int *y) {
5     *y = x + 3;
6   }
7
8   int main () {
9     int a, b;
10
11
12    a = 5; b = 3;
13
14    g (a, &b);
15
16    printf ("a = %d, b = %d\n", a, b);
17
18    return 0;
19  }
```

Note que a função `g` é do tipo **void**. Ela de fato não tem a palavra **return** no final dela. Diferentemente da função `f` do exemplo anterior.

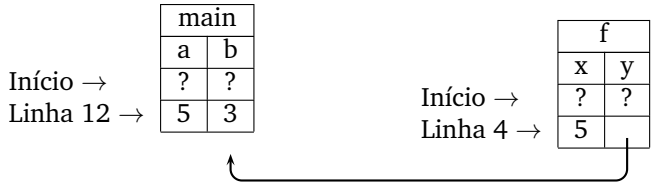
O programa sempre começa a ser executado na função principal.

	main	
	a	b
Início →	?	?
Linha 12 →	5	3

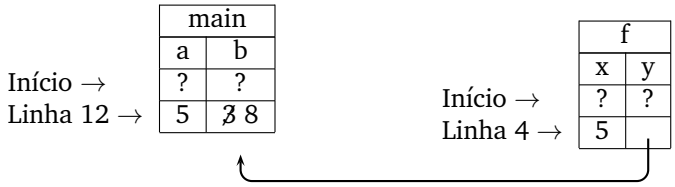
Na Linha 14 é chamada a função `g`. Note a passagem de parâmetros:

- `a` da função `main` → (para) `x` da função `g` e
- `&b` da função `main` → (para) `y` da função `g`

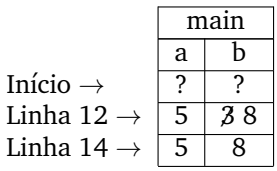
Note que no segundo parâmetro, o parâmetro `y` da função `g` é um ponteiro para **int**. Note que o parâmetro `y` recebe o endereço da variável `b` da função `main`. Dessa forma, o parâmetro `y` da função `f` aponta para a variável `b` da função `main`.



Na Linha 5, estamos usando o ponteiro `y` da função `g` com o operador "vai para". Ou seja, "vai para" a gaveta em que o `y` está apontando e guarde o resultado da expressão `x+3` neste lugar. Como `y` está apontando para a variável `b` da função `main`, então o resultado da expressão `x+3` será guardado na variável `b` da função `main`. Depois da execução da Linha 5, temos então:



Na Linha 6, acabou a função `g` e fluxo do programa volta para onde a função `g` foi chamada, ou seja, na Linha 14.



No final, o programa imprime na tela o contéudo das variáveis `a` e `b` da função `main`:

Tela
a = 5, b = 8

Então, para que serve uma função **void**? Funções do tipo **void** são úteis quando não necessitamos devolver um valor via **return**. E quando isso acontece?

17.8 Exercícios sugeridos

(a) Faça uma função que converte uma coordenada cartesiana (`x`,`y`) em coordenadas polares (`r`,`s`). Esta função tem duas entradas (via parâmetros) que são os valores `x` e `y` e deve devolver dois valores: as coordenadas polares `r` e `s`.

```c
# include <stdio.h>
# include <math.h>

void converte_polar (float x, float y, float *apt_r, float * apt_s) {
  float s;

  *apt_r = sqrt (x*x + y*y);

  /* if (x == 0 && y >0) */
  s = 3.1415 / 2;
  if (x == 0 && y < 0)
  s = 3 * 3.1415 / 2;
  if (x == 0 && y == 0)
  s = 0;
  if (x != 0)
  s = atan (y/x);
  if (x < 0)
  s += 3.1415;
  *apt_s = s;
}
```

(b) Faça um programa que leia um inteiro `n>0` e n coordenadas cartesianas reais (x,y) e imprima as respectivas coordenadas polares.

```c
int main () {
  float x, y, r, s;
  int i, n;

  printf ("Entre com um inteiro n > 0: ");
  scanf ("%d", &n);

  for (i=0; i<n; i++) {
    printf ("Entre com a coordenada x: ");
    scanf ("%f", &x);

    printf ("Entre com a coordenada y: ");
    scanf ("%f", &y);

    converte_polar (x, y, &r, &s);

    printf ("Coordenada (%f, %f)\n", r, s);
  }

  return 0;
}
```

(c) Dizemos que um número natural n é palíndromo se lido da direita para a esquerda ou da esquerda para a direita é o mesmo número. Exemplos:

- 567765 e 32423 são palíndromos.
- 567675 não é palíndromo.
- Socorram-me subi no onibus em Marrocos.
- Oto come doce seco de mocotó.
- A diva em Argel alegra-me a vida.

1) Escreva uma função que recebe um inteiro $n > 0$ e devolve o seu primeiro dígito, seu último dígito e altera o valor de n removendo seu primeiro e último dígitos. Exemplos:

valor inicial de n	primeiro dígito	último dígito	valor final de n
732	7	2	3
14738	1	8	473
1010	1	0	1
78	7	8	0
7	7	7	0

2) Escreva um programa que recebe um inteiro positivo n e verifica se n é palíndromo. Suponha que n não contém o dígito 0.

(d) Escreva um programa que lê dois inteiros positivos m e n e calcula o mínimo múltiplo comum entre m e n. Para isso, escreve primeiro uma função com protótipo:

```
int divide (int *m, int *n, int d);
```

que recebe três inteiros positivos como parâmetros e retorna 1 se d divide pelo menos um entre $*m$ e $*n$, 0 caso contrário. Fora isso, se d divide $*m$, divide $*m$ por d, e o mesmo para $*n$.

(e) Soma de números binários.
1) Faça uma função com protótipo

```
void somabit (int b1, int b2, int *vaium, int *soma);
```

que recebe três bits (inteiros 0 ou 1) $b1$, $b2$ e $*vaium$ e retorna um bit soma representando a soma dos três e o novo um bit ”vai-um”em $*vaium$.

2) Escreva um programa que lê dois números em binário e calcula um número em binário que é a soma dos dois números dados. Utilize a função do item a.

18 Vetores

Ronaldo F. Hashimoto e Carlos H. Morimoto

Nessa aula vamos introduzir o tipo **vetor**. Ao final dessa aula você deverá saber:

- Descrever o que são *vetores* na linguagem C.
- Declarar vetores.
- Como acessar elementos de um vetor e percorrer um vetor.
- Utilizar vetores para resolver problemas computacionais.

18.1 Vetores

Vetores são estruturas indexadas utilizadas para armazenar dados de um mesmo tipo: **int**, **char**, **float** ou **double**. O exemplo a seguir é de um vetor de inteiros:

0	1	2	3	4	5		78	79
10	12	17	12	-4	-3	···	-42	34

18.1.1 Declaração de Vetores

A declaração de um vetor é feita da seguinte forma:

`<tipo_do_vetor> <nome_do_vetor> [<tamanho_do_vetor>];`

Exemplos:

- **int** `v[80];` ——— `80 é o tamanho do vetor!`

  A declaração acima reserva 80 gavetas consecutivas na memória, que corresponde ao tamanho ou número de casas do vetor. Cada gaveta guarda um **int**.

- **float** `x[20];` ——— `20 é o tamanho do vetor!`

  A declaração acima reserva 20 gavetas consecutivas na memória. Cada gaveta guarda um **float**.

Observação Importante:

1. Na **declaração de vetor**, o que está entre colchetes deve ser um **número constante**.
2. Assim, não é possível fazer algo deste tipo:

```
int n = 20;
float x[n]; /* não é permitido declarar colocando uma variável */
```

ou

```
int n;

printf ("Entre com n>0: ");
scanf ("%d", &n);

float x[n];
```

O correto seria:

```
int n;
float x[20]; /* o correto é declarar sempre um tamanho fixo */
```

18.1.2 Uso de Vetores

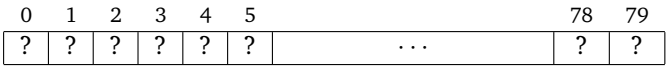
- São usados índices para acessar uma casa de um vetor.
- Um índice é um número natural.
- O índice da **primeira** casa é sempre **zero**.

18.1.3 Exemplo de Uso de Vetores

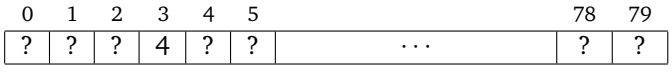
- Exemplo 1:

```
1   # include <stdio.h>
2
3   int main () {
4     int v[80], i;
5
6     v[3] = 4; /* casa de índice 3 do vetor v recebe o inteiro 4 */
7     i = 2;
8     v[i] = 3; /* casa de índice 2 do vetor v recebe o inteiro 3 */
9     v[v[v[i]]] = 10; /* vc saberia dizer qual casa do vetor v
10                      * recebe o inteiro 10?
11                      */
12
13    return 0;
14  }
```

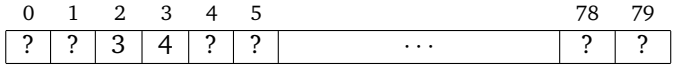
Na Linha 4, o vetor `v` com 80 casas é declarado:



Na Linha 6, casa de índice 3 do vetor v recebe o inteiro 4:



Na Linha 8, casa de índice 2 do vetor v recebe o inteiro 3:



Na Linha 9, temos: `i`=2, `v[i]`=3 e `v[v[i]]`=`v[3]`=4. Desta forma, no comando da Linha 9, a casa de índice 4 do vetor v recebe o inteiro 10:

- Exemplo 2:

```
1   # include <stdio.h>
2
3   int main () {
4     float x[80];
5     int i;
6
7     for (i=0; i<80; i++)
8       x[i] = 0;
9
10    return 0;
11  }
```

O programa acima coloca o valor zero em cada uma das casas do vetor `x`.

- Exemplo 3:

  O índice do vetor pode ser uma expressão aritmética, como mostrado a seguir:

```
1   # include <stdio.h>
2
3   int main () {
4     float x[80];
5     int i;
6
7     for (i=110; i<190; i++)
8       x[i-110] = 0;
9
10    return 0;
11  }
```

  mas tenha absoluta certeza, porém, de sempre fornecer um índice válido de forma que o resultado da expressão aritmética seja válida (neste exemplo, o resultado da expressão aritmética deve ser um inteiro entre 0 e 79).

18.2 Percorrimento de Vetores

Percorrer um vetor significa varrer o vetor de casa em casa a partir do índice 0 (zero). No percorrimento de um vetor, é necessário saber o número de casas que deve-se fazer este percorrimento. Este número normalmente é guardado em uma variável inteira.

Muitos problemas computacionais que envolvem vetores têm como solução o uso de um padrão para percorrimento de vetores.

Um padrão para percorrer "`n_casas`" de um vetor "`vetor`" é usar um comando de repetição (no caso, vamos usar o comando **for**) com uma variável inteira "`indice`" para o índice das casas do vetor:

```
for (indice=0; indice < n_casas; indice++) {
  <algum comando usando vetor[indice]>
}
```

18.2.1 Leitura de um Vetor

Para leitura de vetor, devemos ler elemento a elemento usando o padrão de percorrimento.

```c
# include <stdio.h>

int main () {
  float v[100];
  int i, n;

  printf ("Entre com 0<n<=100: ");
  scanf ("%d" &n);

  /* percorrer o vetor v de 0 a n-1 colocando o valor lido pelo teclado */
  for (i=0; i<n; i++) {
    printf ("Entre com v[%d] = ", i);
    scanf ("%f", &v[i]);
  }

  return 0;
}
```

Observe com cuidado a linha do programa utilizada para ler o vetor:

```c
scanf ("%f", &v[i]);
```

A posição `i` do vetor `v`, ou seja, `v[i]`, é utilizada da mesma forma que utizamos qualquer variável até o momento. Essa "variável" é passada para a funcao `scanf` precedida pelo caractere '&'.

18.2.2 Impressão de um Vetor

Para impressão de vetor, devemos imprimir elemento a elemento usando o padrão de percorrimento.

```c
# include <stdio.h>

int main () {
  float v[100];
  int i, n;

  printf ("Entre com 0<n<=100: ");
  scanf ("%d" &n);

  /* percorrer o vetor v de 0 a n-1 imprimindo o valor de cada casa */
  for (i=0; i<n; i++) {
    printf ("v[%d] = %f\n", i, v[i]);
  }

  return 0;
}
```

18.2.3 Observação sobre Percorrimento

Na declaração de um vetor é definido um número fixo de casas, uma vez que sempre deve-se colocar uma constante na definição do número de casas do vetor. Por exemplo:

```c
int v[30];
```

Mas, como podemos ver nos exemplos de percorrimento para ler e imprimir um vetor, um usuário não necessariamente irá usar todas as casas disponíveis do vetor. Note que no padrão de percorrimento deve sempre

existir uma variável indicando quantas casas do vetor estão sendo verdadeiramente usadas (variável `n_casas` do padrão).

Assim, normalmente, em problemas computacionais que envolvem vetores deve-se sempre ter uma variável inteira associada a este vetor que diz quantas casas do vetor estão sendo usadas (por exemplo, variável inteira `n` associada ao vetor `v` nos exemplos de leitura e impressão de vetores).

18.3 Exercícios Comentados

18.3.1 Exercício 1

Dada uma sequência de `0<n<100` números inteiros, imprimi-la na ordem inversa à da leitura.

Exemplo:

Para `n=5`, e a sequência $11, 12, 3, 41, 321$, o programa deve imprimir a saída $321, 41, 3, 12, 11$.

Para resolver esse problema, precisamos armazenar todos os elementos da sequência em um vetor (usando padrão de percorrimento), e depois imprimir esses elementos em ordem inversa (usando padrão de percorrimento em ordem inversa). Observe que sem usar vetor, ou seja, usando apenas variáveis, seria muito difícil resolver esse problema para um valor arbitrário de `n`. Um programa possível, usando vetores, seria:

```
1       # include <stdio.h>
2
3       # define MAX 100
4
5       int main () {
6         int v[MAX], n, i;
7
8         printf ("Entre com 0<n<100: ");
9         scanf ("%d", &n);
10
11        /* percorrer o vetor v do índice 0 a n-1 colocando o valor lido pelo teclado */
12        for (i=0; i<n; i++) {
13          printf ("Entre com v[%d] = ", i);
14          scanf ("%d", &v[i]);
15        }
16
17        /* percorrer o vetor v do índice n-1 a 0 imprimindo o valor de cada casa */
18        for (i=n-1; i>=0; i--) {
19          printf ("v[%d] = %d\n", i, v[i]);
20        }
21
22        return 0;
23      }
```

Note que neste exercício definimos uma constante `MAX` usando o "comando" `define`. Observe que `MAX` é uma constante e não uma variável. Mais sobre definição de constantes, veja o material didático **Alguns Detalhes da Linguagem C**.

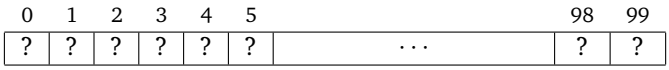
Observe então que o tamanho do vetor é fixo, e deve ser definido antes do programa ser executado. É um erro muito comum entre programadores inexperientes ler um número e utilizá-lo para definir o tamanho do vetor. Algo do tipo:

```
int n;

printf ("Entre com n>0: ");
scanf ("%d", &n);

int v[n];
```

Isto está completamente **ERRADO**! Tome cuidado para você não cometer este **ERRO**!

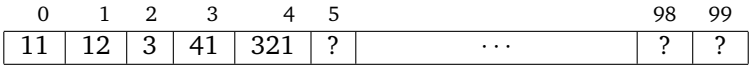
Na solução do problema, no programa, o valor limite foi definido como `MAX` e possui valor 100. Esse programa não pode ser executado para sequências maiores que 100.

Na Linha 6, o vetor `v` com 100 casas é declarado:



A leitura do inteiro `n` (digamos que leia 5, como no exemplo dado no exercício) que indica quantas das 100 casas do vetor `v` que serão de fato utilizadas é feita na Linha 9.

Após a leitura do vetor (Linhas 11 a 15), a estrutura correspondente na memória pode ser representada como:



Note que as casas de índice maior que 4 contêm lixo. Mas, tudo bem, uma vez sabemos o número de casas (guardado na variável `n`=5) que contêm os valores da sequência.

Agora observe a parte do programa para imprimir o vetor invertido (Linhas 17 a 20). Como o vetor foi lido como mostra o esquema acima, devemos percorrer o vetor `v` desde a última posição lida e decrementar até a primeira. A última posição lida nesse caso foi a posição 4 (quatro), e a primeira foi 0 (zero).

18.3.2 Exercício 2

Dados `n`>0 lançamentos de uma roleta (números entre 0 e 36), calcular a frequência de cada número.

Como a gente calcula a frequência de um número? Basta contar quantas vezes ele aparece na sequência. Precisamos, portanto, criar um contador para cada número da sequência possível, ou seja, 37 contadores. Cada contador começa com zero, e toda vez que lancamos a roleta, a gente incrementa apenas o contador correspondente. Ao final, basta calcular as frequências. A solução completa e dada a seguir:

```c
#include <stdio.h>

int main () {
  int freq[37];
  int n, i, x;

  /* zerando o vetor freq */
  for (i=0; i<37; i++)
  freq[i] = 0;

  printf ("Entre com o número de lançamentos n>0: ");
  scanf ("%d", &n);

  for (i=0; i<n; i++) {
    printf ("Entre com um lançamento: ");
    scanf ("%d", &x);
    freq[x] = freq[x] + 1;
  }

  for (i=0; i<37; i++)
    if (freq[i] > 0)
      printf ("O número %d apareceu %d veze(s)\n", i, freq[i]);

  return 0;
}
```

18.3.3 Exercício 3

Um exemplo de como vetores pode ser útil. Um vetor poderia guardar os coeficientes de um polinômio de grau n. Por exemplo:

```c
float a[101];
int n;
```

Dessa forma, $a[0], a[1], \ldots, a[n]$ guardam os coeficientes de um polinômio $p(x) = a[0] + a[1]x + ... + a[n]x^n$. É claro que, neste particular caso, n não pode ser maior que 100.

18.4 Erros Comuns

Ao desenvolver seus programas com vetores, preste atenção com relação aos seguintes detalhes:

- **índices inválidos:** tome muito cuidado, especialmente dentro de um **while** ou **for**, de não utilizar índices negativos ou maiores que o tamanho máximo do vetor.
- **Definição do tamanho do vetor** se faz na declaração do vetor. O tamanho dele é constante, só mudando a sua declaração é que podemos alterar o seu tamanho. Isso significa que podemos estar "desperdiçando" algum espaço da memória que fica no final do vetor. Não cometa o erro de ler `n`, onde `n` seria o tamanho do vetor, e tentar "declarar" o vetor em seguida.

18.5 Exercícios Recomendados

1. Dados dois polinômios reais $p(x) = a_0 + a_1 \cdot x + ... + a_n \cdot x^n$ ($n < 20$) e $q(x) = b_0 + b_1 \cdot x + ... + b_m \cdot x^m$ ($m < 40$) determinar o produto desses polinômios.
2. Dados dois vetores x e y, ambos com n elementos, $n < 50$, determinar o produto escalar desses vetores.

3. Em uma classe há n alunos, $n < 100$, cada um dos quais realizou k provas, $k < 4$, com pesos distintos. Dados n , k, os pesos das k provas e as notas de cada aluno, calcular a média ponderada das provas para cada aluno e a média aritmética da classe em cada uma das provas.

4. Dada uma sequência de 0<n<200 números inteiros, determinar um segmento de soma máxima.

   Exemplo: para n=12 e a sequência

0	1	2	3	4	5	6	7	8	9	10	11	12	···	198	199
5	2	-2	-7	3	14	10	-3	9	-6	4	1	?	···	?	?

   um segmento de soma máxima está entre os índices 4 e 8 e a correspondente soma é 33.

5. Dada uma sequência $x_0, x_1, ..., x_{k-1}$ de números inteiros, com $k < 300$, verifique se existem dois segmentos consecutivos iguais nesta sequência, isto é, se existem i e m tais que:

$$x_i, x_{i+1}, ..., x_{i+m-1} = x_{i+m}, x_{i+m+1}, ..., x_{i+2m-1}$$

   Imprima, caso existam, os valores de i e m.

   Exemplo: para $k = 8$ e a sequência

0	1	2	3	4	5	6	7	8	···	298	299
7	9	5	4	5	4	8	6	?	···	?	?

   existem i=2 e m=2.

6. Dada uma sequência de $n > 0$ números reais, imprimi-los eliminando as repetições.

19 Vetores, Ponteiros e Funções

Ronaldo F. Hashimoto e Carlos H. Morimoto

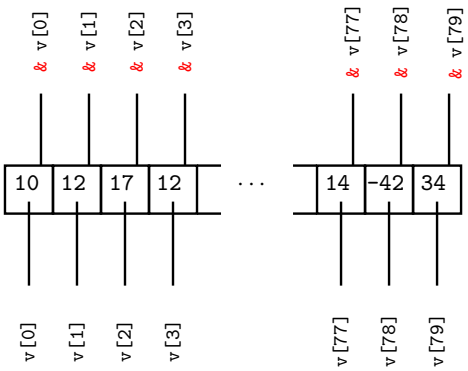
O objetivo desta aula é relacionar o tipo **vetor** com **ponteiros** e assim entender como utilizar vetores como parâmetros de funções. Ao final dessa aula você deverá saber:

- Descrever como os vetores são armazenados na memória.
- Descrever a relação entre vetores e ponteiros.
- Utilizar vetores como parâmetros de funções.

19.1 Vetores

Vimos na aula anterior que **vetores** são estruturas indexadas utilizadas para armazenar dados de um mesmo tipo: **int**, **char**, **float** ou **double**. Por exemplo, a declaração

```
int v[80]; /* declara um vetor de inteiros de nome v com 80 casas */
```

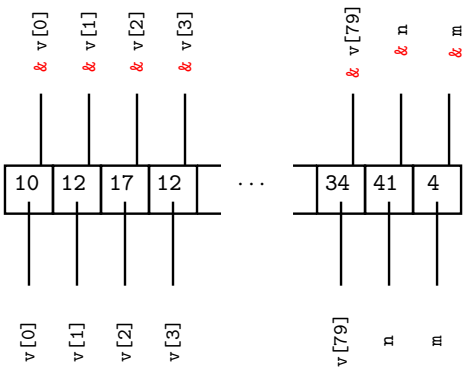


Cada casa do vetor `v` (ou seja, `v[0]`, `v[1]`, ..., `v[79]`) é um inteiro. Além disso, cada casa tem um endereço associado (ou seja, `&v[0]`, `&v[1]`, ..., `&v[79]`).

Uma pergunta que poderíamos fazer é como um vetor fica armazenado na memória. A organização das variáveis na memória depende de como o sistema operacional faz gerenciamento da memória. Em geral, para ser mais eficiente, o sistema operacional tende a colocar as variáveis sucessivamente. Assim, a alocação do vetor na memória é feita de forma sucessiva, ou seja, da maneira como ilustrada na figura acima: `v[0]` antes de `v[1]`, que por sua vez antes de `v[2]` e assim por diante. Assim, as variáveis declaradas como

```
int v[80]; /* declara um vetor de inteiros de nome v com 80 casas */
int n, m;
```

poderiam ser alocadas de forma sucessiva como



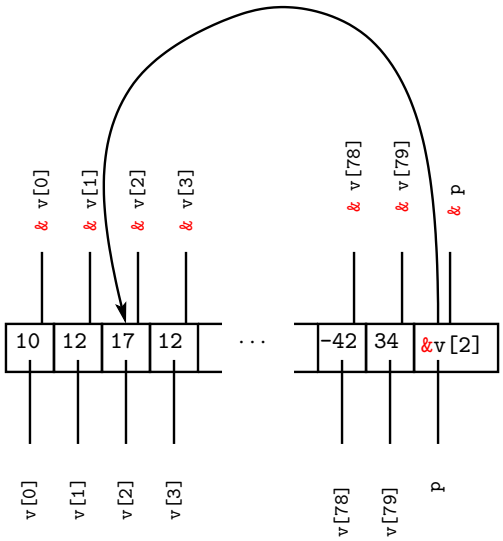
Na linguagem C não existe verificação de índices fora do vetor. Quem deve controlar o uso correto dos índices é o programador. Além disso, o acesso utilizando um índice errado pode ocasionar o acesso de outra variável na memória. No exemplo acima, `v[80]` acessaria a variável `n`. Se o acesso à memória é indevido você recebe a mensagem "segmentation fault".

19.2 Vetores e Ponteiros

A implementação de vetores em C está bastante interligada com a de ponteiros visando a facilitar a manipulação de vetores. Considere a seguinte declaração de variáveis:

```
int v[80]; /* declara um vetor de inteiros de nome v com 80 casas */
int *p;
```

que aloca na memória algo do tipo:



Podemos utilizar a sintaxe normal para fazer um ponteiro apontar para uma casa do vetor:

```
p = &v[2]; /* p aponta para a casa de índice 2 de v */
```

Mas podemos utilizar a **sintaxe especial para ponteiros e vetores**, junto com as operações para ponteiros:

- Podemos fazer um ponteiro apontar para o início do vetor `v` fazendo

  ```
  p = v;
  ```

  É a única situação em que o nome do vetor tem sentido sem os colchetes. O comando acima equivale a fazer `p = &v[0];`

- Podemos usar a sintaxe de vetores (`nome_do_vetor[índice]`) com o ponteiro. Assim, se fizermos

  ```
  p = v;
  ```

  podemos acessar o elemento que está na casa `i` de `v` fazendo `p[i]`, ou seja, ambos `p[i]` e `v[i]` acessam a casa `i` do vetor `v`. Exemplo:

  ```
  i = 3;
  p = v; /* p aponta para v[0]. Equivale a fazer p = &v[0] */
  p[i] = 4; /* equivale a fazer v[i] = 4 */
  ```

  Mas se fazemos

  ```
  p = &v[3];
  ```

  então, `p[0]` é o elemento `v[3]`, `p[1]` é o elemento `v[4]`, `p[2]` é o elemento `v[5]`, e assim por diante.

- Podemos fazer algumas operações com ponteiros. Considere a seguinte declaração:

  ```
  int *p, *q, n, v[50];
  float *x, y[20];
  ```

  1. Quando somamos 1 a um ponteiro para **int** (por exemplo, `p`) ele passa a apontar para o endereço de memória logo após a memória reservada para este inteiro. Exemplo, se `p = &v[4]`, então `p+1` é o endereço de `v[5]`, `p+2` é o endereço de `v[6]`, `p+i` é o endereço de `v[4+i]`.
     Dessa forma, `*p` (vai para onde o `p` está apontando) é o `v[4]`. Portanto, `v[4] = 3` é a mesma coisa que fazer `*p = 3`. Como `p+1` é o endereço de `v[5]`, então `*(p+1)` é `v[5]`. Assim, `v[5] = 10` é a mesma coisa que fazer `*(p+1) = 10`.
  2. Se somamos 1 a um ponteiro para **float** (por exemplo `x`) ele avança para o endereço após este **float**. Por exemplo, se `x=&y[3]`, então `x+1` é o endereço de `y[4]` e `x+i` é o endereço de `y[3+i]`.
  3. Somar ou subtrair um inteiro de um ponteiro:

     ```
     p = &v[22]; q = &v[30];
     p = p - 4; q++;
     *(p+2) = 3; *q = 4;
     ```

     Qual índice de `v` recebe 3? Qual índice de `v` recebe 4? [5]
  4. Subtrair dois ponteiros:

     ```
     p = &v[20]; q = &v[31];
     n = q - p; /* número inteiro: a diferença entre os índices, neste caso,
     ```

[5] As respostas são: `v[20]` recebe 3 e `v[31]` recebe 4.

19.3 Vetores como Parâmetro de Funções

Quando se declara uma função que tem como parâmetro um vetor, este vetor é declarado somente com abre e fecha colchetes. Exemplo:

```
# include <math.h>
float modulo (float v[], int n) {
  int i;
  float r = 0;
  for (i=0; i<n; i++) {
    r = r + v[i]*v[i];
  }
  r = sqrt (r);
  return r;
}
```

Esta função recebe um vetor de reais `v` com `n` elementos e devolve o seu módulo via **return**. A declaração acima é equivalente a

```
float modulo (float *p , int n) {
  int i;
  float r = 0;
  for (i=0; i<n; i++) {
    r = r + p[i]*p[i];
  }
  r = sqrt (r);
  return r;
}
```

Na verdade, a declaração no argumento **float** `v[]` é a mesma coisa que **float** `*p`, ou seja, `v` é um ponteiro.

```
1  # include <stdio.h>
2  # include <math.h>
3
4
5  float modulo (float v[], int n) {
6    int i;
7    float r = 0;
8    for (i=0; i<n; i++) {
9      r = r + v[i]*v[i];
10   }
11   r = sqrt (r);
12   return r;
13 }
14
15 int main () {
16   float x[100], comprimento;
17   int m;
18
19   m = 3;
20   x[0] = 2; x[1] = -3, x[2] = 4;
21
22
23   comprimento =  modulo (x, m);
24
25   printf ("Comprimento = %f\n", comprimento);
26
27   return 0;
28 }
```

O parâmetro `v` da função `modulo` aponta para a variável `x[0]` da função `main`. Então `v[i]` na função `modulo` é exatamente `x[i]` da função `main`.

19.4 Exemplo de Função com Vetor como Parâmetro

O nome de um vetor dentro de parâmetro de uma função é utilizado como sendo um ponteiro para o primeiro elemento do vetor na hora de utilizar a função.

Exemplo de declaração de funções com vetores como parâmetros:

```
1       # define MAX 200
2
3
4       float f (float u[]) {
5         float s;
6         /* declaração da função f */
7         ...
8         u[i] = 4;
9         ...
10        return s;
11      }
12
13      int main () {
14        float a, v[MAX]; /* declaração da variável a e vetor v */
15        ...
16        /* outras coisas do programa */
17
18
19        a = f (v);   /* observe que o vetor é passado apenas pelo nome */
20
21        ...
22
23        return 0;
24      }
```

u aponta para v[0].

Na Linha 19, a chamada da função `f` faz com que o ponteiro `u` receba `&v[0]`, ou seja, faz com que o ponteiro `u` aponte para `v[0]`.

Na Linha 8, temos o comando `u[i] = 4`. Como `u` está apontando para `v[0]`, então o comando `u[i] = 4` é o mesmo que fazer `v[i] = 4`. Assim, na Linha 8, dentro da função `f`, estamos mudando o conteúdo da casa de índice `i` do vetor `v` da função `main`.

19.5 Problema

(a) Faça uma função que recebe dois vetores de tamanho n e retorna o seu produto escalar.

O protótipo dessa função seria:

```
float ProdutoEscalar (float u[], float v[], int n);
```

A função recebe como parâmetros os vetores u e v, e um inteiro n. Uma possível solução para esta função seria:

```
float ProdutoEscalar (float u[], float v[], int n) {
  int i;
  float res = 0;
  for (i=0; i<n; i++)
  res = res + u[i] * v[i];
  return res;
}
```

(b) Faça um programa que leia dois vetores reais de tamanho $n < 200$ e verifique se eles são vetores ortogonais. Dois vetores são ortogonais se o produto escalar entre eles é zero. Considere EPS igual a 0.001 o valor do erro para comparar se o produto escalar é zero.

```
#include <stdio.h>

#define MAX 200
#define EPS 0.001

float ProdutoEscalar (float u[], float v[], int n) {
  int i;
  float res = 0;
  for (i=0; i<n; i++)
  res = res + u[i] * v[i];
  return res;
}

int main () {
  int n, i;
  float a[MAX], b[MAX];
  float prod;

  /* leitura dos vetores */
  printf("Digite o tamanho dos vetores: ");
  scanf("%d", &n);

  printf("Entre com os valores do 1o vetor\n");
  for (i=0; i<n; i++) scanf("%f",&a[i]);

  printf("Entre com os valores do 2o vetor\n");
  for (i=0; i<n; i++) scanf("%f",&b[i]);

  prod = ProdutoEscalar(a, b, n);

  /* cuidado com a comparação com zero usando reais!!! */
  if (prod < EPS && prod > -EPS)
    printf("Os vetores sao ortogonais.\n");
  else
    printf("Os vetores nao sao ortogonais.\n");

  return 0;
}
```

Observe que a função `ProdutoEscalar` não modifica o vetor u nem o vetor v.

Agora, vamos considerar o caso quando temos uma função que deve retornar um vetor.

(c) Faça uma função que recebe dois vetores de tamanho 3 e retorna o seu produto vetorial. O produto vetorial de dois vetores de dimensão três $u = (u_0, u_1, u_2)$ e $v = (v_0, v_1, v_2)$ é dado por $w = (u_1v_2 - u_2v_1, u_2v_0 - u_0v_2, u_0v_1 - u_1v_0)$.

Primeiro como deve ser o protótipo dessa função? Nós sabemos que a função deve receber 2 vetores de entrada e devolver 1 vetor como resultado. Sendo assim, temos o seguinte protótipo:

```
void ProdutoVetorialTRI (float u[], float v[], float w[]);
```

onde os vetores u e v são entradas e w é o vetor de saída. Uma solução para esta função possível seria:

```
void ProdutoVetorialTRI (float u[], float v[], float w[]) {
  w[0] = u[1]*v[2] - u[2]*v[1];
  w[1] = u[2]*v[0] - u[0]*v[2];
  w[2] = u[0]*v[1] - u[1]*v[0];
}
```

Observe que a função `ProdutoVetorialTRI` modifica o vetor w. Este vetor w é na verdade um ponteiro para algum vetor da função `main`. E é este vetor que na realidade vai ser modificado.

(d) Faça um programa que leia dois vetores de dimensão três e calcula o seu produto vetorial, e mostra que o produto vetorial é ortogonal aos dois vetores de entrada.

```c
#include <stdio.h>

#define MAX 20

float ProdutoEscalar (float u[], float v[], int n) {
  int i;
  float res = 0;
  for (i=0; i<n; i++)
  res = res + u[i] * v[i];
  return res;
}

void ProdutoVetorialTRI (float u[], float v[], float w[]) {
  w[0] = u[1]*v[2] - u[2]*v[1];
  w[1] = u[2]*v[0] - u[0]*v[2];
  w[2] = u[0]*v[1] - u[1]*v[0];
}

int main () {
  int n, i;
  float a[MAX], b[MAX], c[MAX];
  float prod;

  n = 3; /* os vetores têm dimensão 3 */

  printf("Entre com os valores do 1o vetor\n");
  for (i=0; i<n; i++) scanf("%f",&a[i]);

  printf("Entre com os valores do 2o vetor\n");
  for (i=0; i<n; i++) scanf("%f",&b[i]);

  /* Observe a chamada da função ProdutoVetorialTRI     */
  /* O produto vetorial de a e b é colocado no vetor c */
  /* via ponteiro w da função ProdutoVetorialTRI        */
  ProdutoVetorialTRI (a, b, c);

  printf ("Produto vetorial (a x b) = (%.2f, %.2f, %.2f)\n", c[0], c[1], c[2]);

  prod = ProdutoEscalar (a, b, n);
  printf("Produto escalar de a e b: %.2f \n", prod);

  prod = ProdutoEscalar (a, c, n);
  printf("Produto escalar de a e c: %.2f \n", prod);

  prod = ProdutoEscalar (b, c, n);
  printf("Produto escalar de b e c: %.2f \n", prod);

  return 0;
}
```

Observe que a função `ProdutoVetorialTRI` modifica o vetor w. Este vetor w é na verdade um ponteiro para o vetor c da função `main`. E é este vetor que na realidade vai ser modificado, ou seja, o produto vetorial fica armazenado no vetor c.

19.6 Outro Problema

(a) Faça uma função que recebe um inteiro $n > 0$ e um vetor de números reais a (que armazena os coeficientes de um polinômio $p(x) = a_0 + a_1 \cdot x + \ldots + a_n \cdot x^n$ de grau n) e devolve a derivada de $p(x)$ no próprio vetor a. Além disso, devolve via **return** o grau do polinômio derivada.

```
int derivada (int n, float a[]) {
  int i;
  for (i=0; i<n;i++) {
    p[i] = (i+1) * a[i+1];
  }
  return n - 1;
}
```

(b) Faça uma função que recebe um inteiro $n > 0$, um vetor de números reais a (que armazena os coeficientes de um polinômio $p(x) = a_0 + a_1 \cdot x + \ldots + a_n \cdot x^n$ de grau n) e um real y devolve $p(y)$, ou seja, o valor do polinômio no ponto y.

```
float valor (int n, float a[], float y) {
  float soma = 0, poty = 1;
  int i;
  for (i=0; i<=n; i++) {
    soma = soma + a[i] * poty;
    poty = poty * y;
  }
  return soma;
}
```

(c) Faça um programa que leia um inteiro $m > 0$, os coeficientes reais de um polinômio $p(x) = a_0 + a_1 \cdot x + \ldots + a_m \cdot x^m$ de grau m e um real y e imprime $p'(p(y) - 2) + p''(y + 2)$.

```
int main () {
  float a[200], y, r, s;
  int m, i;

  printf ("Entre com o grau do polinomio: ");
  scanf ("%d", &m);

  printf ("Entre com os coeficientes do polinomio\n");
  for (i=0; i<=m; i++) {
    printf ("Entre com o coeficiente a[%d] = ");
    scanf ("%f", &a[i]);
  }

  printf ("Entre com o ponto y: ");
  scanf ("%f", &y);

  /* calculando p(y) */
  r = valor (m, a, y);

  /* calculando a derivada de p(x) */
  m = derivada (m, a);

  /* calculando p'(r-2) */
  r = valor (m, a, r-2);

  /* calculando a derivada de p'(x) */
  m = derivada (m, a);

  /* calculando p''(y+2) */
  s = valor (m, a, y+2);

  /* imprimindo resposta final */
  printf ("resposta = %f\n", r+s)

  return 0;
}
```

19.7 Observação

É possível também declarar o tamanho `MAX` do vetor nos parâmetros de função, por exemplo, da seguinte forma:

```
float ProdutoEscalar (float u[MAX], float v[MAX], int n);
void ProdutoVetorialTRI (float u[MAX], float v[MAX], float w[MAX]);
```

Note que o tamanho do vetor é irrelevante na definição da função, no entanto, alguns programadores preferem colocar explicitamente o tamanho `MAX` dos vetores.

19.8 Resumo

Vetor quando passado como parâmetro de função é um **ponteiro**!

20 Caracteres - Tipo char

Ronaldo F. Hashimoto e Carlos H. Morimoto

Até agora vimos como o computador pode ser utilizado para processar informação que pode ser quantificada de forma numérica. No entanto, há muita informação que não é numérica, como por exemplo o seu nome, seu endereço residencial, uma fotografia sua ou o som de sua voz. Textos são compostos por caracteres do alfabeto, de pontuação, acentuação, etc. Para que possam ser processados pelo computador, precisamos de uma forma para representar caracteres utilizando apenas números.

Ao final dessa aula você deverá saber:

- Declarar, ler e imprimir variáveis do tipo `char`.
- Descrever como caracteres são representados no computador e porque eles podem ser utilizados em expressões.
- Utilizar variáveis do tipo `char` em programas.

20.1 Caracteres

Um caractere pode ser uma letra (maiúscula ou minúscula), um ponto final, ponto de interrogação, colchetes, enfim, símbolos que normalmente encontramos num teclado de um computador. Nesta aula, estudaremos as variáveis utilizadas em C para armazenar caracteres.

Em C, caracteres são armazenados como números inteiros, normalmente utilizando uma tabela de conversão amplamente difundida chamada Tabela ASCII (American Standard Code for Information Interchange).

A Tabela ASCII original possui somente 128 caracteres, aqueles com código entre 0 e 127. Veja na tabela abaixo, a correspondência entre caractere e número (entre os números 32 e 127).

```
  032   ! 033   " 034   # 035   $ 036   % 037   & 038   ’ 039
( 040   ) 041   * 042   + 043   , 044   - 045   . 046   / 047
0 048   1 049   2 050   3 051   4 052   5 053   6 054   7 055
8 056   9 057   : 058   ; 059   < 060   = 061   > 062   ? 063
@ 064   A 065   B 066   C 067   D 068   E 069   F 070   G 071
H 072   I 073   J 074   K 075   L 076   M 077   N 078   O 079
P 080   Q 081   R 082   S 083   T 084   U 085   V 086   W 087
X 088   Y 089   Z 090   [ 091   \ 092   ] 093   ^ 094   _ 095
‘ 096   a 097   b 098   c 099   d 100   e 101   f 102   g 103
h 104   i 105   j 106   k 107   l 108   m 109   n 110   o 111
p 112   q 113   r 114   s 115   t 116   u 117   v 118   w 119
x 120   y 121   z 122   { 123   | 124   } 125   ~ 126     127
```

Existem várias extensões da tabela ASCII utilizando também os números negativos de -1 a -128. Normalmente, estas tabelas estendidas incluem os caracteres acentuados.

A cada caractere corresponde a um número entre 0 e 127 e a cada número entre 0 e 127 corresponde a um caractere. Caractere com código 32 corresponde ao espaço em branco e o caractere com código 127 varia de computador a computador.

Os caracteres com código decimal entre 0 e 31 são chamados de caracteres de controle, ou seja, eles indicam alguma coisa ao que a impressora ou monitor de vídeo devem executar. Entre eles, os mais utilizados são:

Caractere	Código ASCII	Significado
nulo	0	este caractere é simplemente ignorado pela impressora ou vídeo.
backspace	8	volta um caractere
line feed	10	pula uma linha
carriage return	13	o cursor vai para a primeira coluna.

Assim, quando você digita a letra `A` (ou o backspace) no teclado, na verdade, o seu computador armazena na memória o número inteiro `65` (ou o número `8`, respectivamente), ou seja, para armazenar a letra `A`, o computador armazena o número `65`.

20.2 Decorar a Tabela ASCII?

Para evitar a necessidade de decorar a tabela ASCII, na linguagem C, escrever um caractere entre apóstrofes equivale a escrever o seu código ASCII. Assim, escrever `65` e `'A'` são equivalentes. Consequentemente, os comandos `i = 65` e `i = 'A'` são equivalentes, assim como `x = x + 65` e `x = x + 'A'`. Dessa forma, observe o comando `A = 'A'`. Neste caso, `A` é uma variável e `'A'` é o caractere cujo código ASCII é `65`; neste comando, o número `65` é armazenado na variável `A`.

Os caracteres são ordenados de acordo com seu código ASCII. Assim o caractere `'0'` (zero) é menor que o caractere `'1'` (um) pois o código ASCII do caractere zero (`48`) é menor que o código ASCII do caractere um (`49`). Assim, `'0' < '1'`, pois `48 < 49`. Da mesma forma, `'1' < '2' < '9' < ... < 'A' < 'B' < ... < 'Z' < ... < 'a' < 'b' < ... < 'z'`, pois `49 < 50 < ... < 65 < 66 < ... < 90 < ... < 97 < 98 < ... < 122`.

20.3 Variável Tipo Char

Uma variável inteira de 32 bits (4 bytes) pode armazenar números inteiros no intervalo de $-2.147.483.648$ a $+2.147.483.647$. No compilador gcc, este tipo de variável é declarado como `int`. Assim, nestes compiladores, a declaração `int x` declara uma variável inteira `x` que armazena inteiros com 4 bytes, ou seja, a variável inteira `x` pode armazenar inteiros compreendidos entre $-2.147.483.648$ a $+2.147.483.647$.

Para armazenar um caractere, poder-se-ia utilizar uma variável inteira (uma vez que um caractere é um número inteiro). No entanto, observe que estaríamos utilizando mais memória (bytes) do que precisaríamos, uma vez que para qualquer caractere, o seu respectivo código ASCII é inteiro entre -128 a $+127$. Uma variável inteira de 8 bits (1 byte) pode armazenar números inteiros compreendidos entre -128 a $+127$, exatamente o que precisaríamos para armazenar o código ASCII de um caractere. Este tipo de variável é declarado como `char`.

A forma para declarar uma variável do tipo `char` é a mesma para declarar variáveis do tipo `int`; só que em vez de usar a palavra chave `int`, deve-se usar a palavra `char`:

char <`nome_da_variavel`>;

Exemplo: declaração de uma variável do tipo `char` de nome `"ch"`

char `ch`;

Se você quiser declarar várias variáveis, é possível fazer da seguinte forma:

char <`nome_da_variavel_1`>, <`nome_da_variavel_2`>, <`nome_da_variavel_3`>,..., <`nome_da_variavel_n`>;

Exemplo: declaração de duas variáveis do tipo `char` `"ch1"` e `"ch2"`.

char `ch1, ch2`;

20.4 Leitura de um Caractere pelo Teclado

Como vimos nas aulas passadas, para ler um número inteiro pelo teclado, nós usamos o "%d" dentro do comando scanf. Assim, para ler um inteiro x fazemos:

```
1       int x;
2
3       printf ("Entre com um numero x > 0: ");
4       scanf  ("%d", &x);
```

Para ler um caractere pelo teclado, você deve utilizar "␣%c" dentro do comando scanf. Aqui temos duas observações. A primeira é o "espaço em branco" antes do %c. A segunda é que, no Linux, a tecla <ENTER> corresponde ao caractere de código ASCII 10; no Windows, a tecla <ENTER> deve gerar uma sequência de dois caracteres: um cujo código ASCII é 13 e outro de código ASCII 10. Para o exemplo, vamos considerar que estamos trabalhando no Linux.

Antes de comentar o "truque" do "espaço em branco" antes do %c, vamos dar uma idéia do que acontece na leitura de um caractere pelo teclado. O comando scanf ("%c", &ch) fica esperando o usuário digitar uma tecla e em seguida um <ENTER>, o programa converte a tecla digitada para o número correspondente ao código ASCII e este número é armazenado na variável inteira de 1 byte de nome ch.

Para mostrar o porque do "espaço em branco" antes do %c dentro do scanf, considere o seguinte trecho de programa que lê dois caracteres usando dois comandos scanf:

```
1       char a, b;
2
3       printf ("Entre com um caractere: ");
4       scanf  ("%c", &a);
5
6       printf ("Entre com um outro caractere: ");
7       scanf  ("%c", &b);
```

Note que neste particular exemplo, não foi utilizado "espaço em branco" antes do %c dentro de cada scanf. Neste trecho de programa, para ler o primeiro caractere, o usuário deverá digitar um caractere do teclado (por exemplo, a letra 'A') e posteriormente dar um <ENTER>. Como o <ENTER> é também um caractere (um caractere cujo código ASCII é 10), ele será lido no segundo scanf (que contém a variável b). Assim, a variável a irá conter o número 65 e a variável b o número 10. Bem, provavelmente, a intenção do programador deste trecho não era ler o <ENTER> na variável b.

Para evitar que o seu programa confunda o <ENTER> como caractere a ser lido, é colocado um "espaço em branco" antes do %c dentro de cada scanf. Assim, o trecho de programa *corrigido* fica:

```
1       char a, b;
2
3       printf ("Entre com um caractere: ");
4       scanf  (" %c", &a);
5
6       printf ("Entre com um outro caractere: ");
7       scanf  (" %c", &b);
```

Só que esta maneira de "enganar" traz consigo um pequeno problema. Além de não ler o caractere <ENTER>, o seu programa não vai ser capaz de ler o caractere "espaço em branco" e o caractere correspondente à tabulação. Mas isso pode não ser incoveniente, uma vez que em muitos problemas de computação envolvendo caracteres, os caracteres de tabulação e "espaço em branco" devem ser mesmo ignorados.

20.5 Impressão de Caracteres

Como vimos nas aulas passadas, para imprimir um número inteiro na tela, nós usamos o "%d" dentro do comando `printf`. Assim, para imprimir um inteiro `x` fazemos:

```
1       int x;
2
3       printf ("Entre com um numero x > 0: ");
4       scanf  ("%d", &x);
5
6       printf ("Número lido foi = %d\n", x);
```

Para imprimir um caractere na tela, nós devemos usar o "%c" dentro do comando `printf`. Note que agora não temos mais o "espaço em branco" antes do `%c` dentro do `printf`. Assim, considere o seguinte trecho de programa que lê dois caracteres ignorando "espaços em branco", <ENTER> e tabulações:

```
1       char a, b;
2
3       printf ("Entre com um caractere: ");
4       scanf  (" %c", &a);
5       printf ("Primeiro Caractere Digitado: %c\n", a);
6
7       printf ("Entre com um outro caractere: ");
8       scanf  (" %c", &b);
9       printf ("Segundo Caractere Digitado: %c\n", b);
```

Eventualmente, poderíamos estar interessados em imprimir o código ASCII do caractere em vez do próprio caractere. Neste caso, basta colocar `%d` no lugar do `%c`. Assim, o seguinte trecho de programa escreve também o código ASCII do caractere digitado:

```
1       char a, b;
2
3       printf ("Entre com um caractere: ");
4       scanf  (" %c", &a);
5       printf ("Primeiro Caractere Digitado: %c (Codigo ASCII = %d)\n",a, a);
6
7       printf ("Entre com um outro caractere: ");
8       scanf  (" %c", &b);
9       printf ("Segundo Caractere Digitado: %c (Codigo ASCII = %d)\n", b, b);
```

20.6 Exemplos de Exercício Envolvendo Caracteres

20.6.1 Exemplo 1

Faça um programa em C que imprima todos os caracteres cujo códigos ASCII estão entre 32 e 126.

Solução:

```
#include <stdio.h>

int main () {
  char c;

  for (c = 32; c < 127; c++)
  printf("caractere %c com ASCII %d\n", c, c);

  return 0;
}
```

Observe que, como o tipo `char` na verdade armazena um número correspondente ao código ASCII do caractere, esse número pode ser impresso como um número inteiro. No `printf` dentro do `for`, a variável `c` é utilizada tanto como inteiro (usando `'%d'` na impressão), como caractere (usando `'%c'` para impressão).

Você pode utilizar variáveis do tipo `char` dentro de expressões numéricas como se fossem variáveis inteiras, porém, lembre-se de que, como uma variável `char` utiliza apenas um byte, os valores que ela pode representar variam de -128 a +127.

Uma solução usando variável inteira é possível. No entanto, há um desperdício de memória, pois uma variável inteira ocupa 4 bytes; enquanto que uma variável do tipo `char` ocupa 1 byte:

```
#include <stdio.h>

int main () {
  int c;

  for (c = 32; c < 127; c++)
  printf("caractere %c com ASCII %d\n", c, c);

  return 0;
}
```

20.6.2 Exemplo 2

Escreva um programa que leia um caractere minúsculo e transforme-o em maiúsculo.

Solução:

para escrever esse programa, não precisamos utilizar a tabela ASCII diretamente, apenas precisamos saber como ela foi definida.

De uma forma geral, é bom saber que os dígitos de `'0'` a `'9'` ocupam posições na tabela antes das letras maiúsculas `'A'` a `'Z'`, que por sua vez ocupam posições na tabela antes das letras minúsculas de `'a'` a `'z'`. Assim, como vimos anteriormente, é possível comparar dois caracteres, de forma que a seguinte relação é válida: `'0' < '1' < ... < '9' < 'A' < 'B' < ... < 'Z' < 'a' < ... < 'z'`.

Agora, se você observar a tabela, os códigos ASCII dos caracteres `'A'` e `'a'` são `65` e `97`, respectivamente. Assim, a diferença entre `'A'` e `'a'` é `32`. O mesmo acontece com os caracteres `'B'` e `'b'` e assim por diante. Dessa forma, para converter um caractere minúsculo para maiúculo, basta subtrair o código ASCII do caractere minúsculo de `32` para se obter o código ASCII do caractere maiúsculo. Assim:

```
char ch;

ch = 'a';

ch = ch - 32;

printf ("Caractere Maiusculo = %c", ch);
```

Na linha 3, a variável `ch` recebe o código ASCII do caractere `'a'` minúsculo, ou seja, o número `97`. Na linha 5, o valor de `ch` é subtraído de `32` ficando com o valor `65`. Na linha 7, o caractere cujo ASCII é `65` é impresso, ou seja, o caractere `'A'`.

Agora, e se você não soubesse que a diferença entre os códigos ASCII entre maiúsculas e minúsculas fosse `32`. E se um dia construirem uma tabela diferente em que esta diferença mudasse? O que a gente poderia fazer é a seguinte conta:

```
1       char dif;
2
3       dif = 'a' - 'A';
```

A variável `dif` guarda o resultado da diferença entre os códigos ASCII dos caracteres `'a'` e `'A'`.

Assim, uma solução para o nosso exercício de conversão de minúscula para maiúscula pode ser:

```
1     #include <stdio.h>
2
3     int main () {
4       char letra, dif;
5       printf ("Digite uma letra: ");
6       scanf (" %c", &letra);
7       dif = ('a' - 'A');
8       if (letra >= 'a' && letra <= 'z') {
9         /* sabemos que eh uma letra minuscula */
10        letra = letra - dif;
11        printf ("Maiuscula: %c\n", letra);
12      }
13      else
14        printf("%c nao e uma letra minuscula\n", letra);
15
16      return 0;
17    }
```

20.7 Exercícios recomendados

1. Dada uma sequência de caracteres terminada por um ponto '.', representando um texto, determinar a frequência relativa de vogais no texto (por exemplo, no texto "Em terra de cego quem tem um olho é caolho", essa frequência é 16/42).
2. Dada uma frase terminada por '.', imprimir o comprimento da palavra mais longa.
3. Dada uma sequência de caracteres terminada por '.', determinar quantas letras minúsculas e maiúsculas aparecem na sequência.
4. Dada uma frase terminada por '.', determinar quantas letras e quantas palavras aparecem no texto. Por exemplo, no texto "O voo GOL547 saiu com 10 passageiros." há 25 letras e 7 palavras.

21 Strings

Ronaldo F. Hashimoto e Carlos H. Morimoto

O objetivo desta aula é introduzir o conceito de *strings*. Ao final dessa aula você deverá saber:

- Descrever o que são *strings*.
- Descrever a distinção entre *strings* e vetores de caracteres.
- Ler e imprimir *strings*.
- Usar recursos da *string.h*.
- Utilizar *strings* em seus programas.

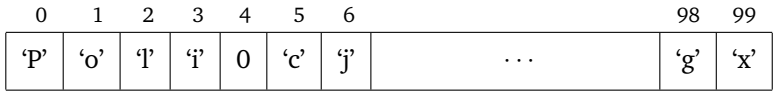
21.1 O que são *strings*?

Na maioria das vezes que temos que programar nos deparamos com a necessidade de uma estrutura de dados que armazena uma sequência de caracteres. Por exemplo, uma mensagem para o usuário, o nome de uma pessoa, seu endereço, seu email e assim por diante. *Strings* não são nada mais do que **vetores de caracteres** com um código para marcar sua terminação. Então, como vimos na aula de vetores, um *string* pode ser então definido da seguinte forma:

char <nome_do_string> [<numero_de_casas>];

Exemplo:

```
char palavra[100]; /* declaração de um vetor de caracteres */
```



O nosso vetor acima poderia guardar uma frase com no máximo 100 caracteres. No nosso exemplo, o vetor `palavra` guarda a sequência de caracteres "Poli". Observe que das 100 casas reservadas, a palavra "Poli" usa somente 4 casas. Assim, precisamos designar o término de uma sequência de caracteres armazenada em um string. Para isso, é reservado um caractere especial para ser usado no vetor ao final da palavra designando assim o seu término. O caractere especial designado para indicar fim da sequência é o caractere de código ASCII 0 (zero) também denotado por `NULL` ou '\0' (barra zero). Obserque que o caractere '\0' não pode fazer parte de nenhuma sequência. Agora, não confunda o caractere '\0' (cujo código ASCII é zero) com o dígito '0' (cujo código ASCII é 48). Note também que todos os caracteres depois do '\0' são considerados lixo dentro do vetor.

21.2 Leitura de Strings

Uma forma de ler um string em C (existem outras formas, mas não serão discutidas aqui) é utilizar o `scanf` com `%[^\n]` e em seguida coloca-se o nome do vetor.

Exemplo:

```
1  char palavra[100]; /* declaração de um vetor de caracteres */
2  printf ("Entre com uma palavra: ");
3  scanf ("%[^\n]", palavra);
```

Na Linha 3, temos um exemplo do uso do `scanf` para leitura de uma sequência de caracteres que vai ser armazenada no vetor (string) `palavra`. O usuário vai digitar uma sequência de caracteres e depois, para terminar, digita a tecla "enter" do teclado, que produz o caractere '\n' de código ASCII 10. O comando `scanf` com %[^\n] lê caractere a caractere e coloca um a um em cada casa do vetor `palavra`; no final, o caractere '\n' de código ASCII 10 é ignorado (ou seja, não é colocado no vetor) e coloca-se o caractere '\0' de código ASCII 0 (zero). Assim, se o usuário digitar

```
MAC<enter>
```

o vetor `palavra` vai conter:

0	1	2	3	4	5	6		98	99
'M'	'A'	'C'	0	0	'c'	'j'	...	'g'	'x'

Note que o número 0 (zero) na casa 3 é o código ASCII do caractere '\0' (proveniente da substituição do caractere '\n' gerado pela tecla "enter" por '\0'). Assim, para representação do string, pode-se optar por colocar o código ASCII (número inteiro zero) ou o caractere entre apóstrofes '\0'. Observe ainda que não é necessário mexer nos outros caracteres depois do primeiro zero , pois eles são considerados como lixo.

21.3 Impressão de Strings

Uma forma de imprimir um string em C (existem outras formas, mas não serão discutidas aqui) é utilizar o `scanf` com %s e em seguida coloca-se o nome do vetor.

```
1    char palavra[100]; /* declaração de um vetor de caracteres */
2    printf ("Entre com uma palavra: ");
3    scanf ("%[^\n]", palavra);
4
5    printf ("A palavra digitada foi: %s\n", palavra);
```

Na Linha 5, temos um exemplo do uso do `scanf` para impressão da sequência de caracteres armazenada no vetor `palavra`.

e uma sequência de caracteres que vai ser armazenada no vetor (string) `palavra`. O usuário vai digitar uma sequência de caracteres e depois, para terminar, digita a tecla "enter" do teclado, que produz o caractere '\n' de código ASCII 10. O comando `scanf` com %[^\n] lê caractere a caractere e coloca um a um em cada casa do vetor `palavra`; no final, o caractere '\n' de código ASCII 10 é ignorado (ou seja, não é colocado no vetor) e coloca-se o caractere '\0' de código ASCII 0 (zero). Assim, se o usuário digitar

21.4 Biblioteca <string.h>

Em C existe uma biblioteca de funções para manipular strings. A seguir, algumas funções:

- **int** `strlen` (**char** `s`[]);

  devolve via **return** o comprimento do string armzenado no vetor `s`.

  Exemplo:

```
#include <stdio.h>
#include <string.h>

int main () {
  ⋮

  char palavra[100]; /* declaração de um vetor de caracteres */
  printf ("Entre com uma palavra: ");
  scanf ("%[^\n]", palavra);

  printf ("%s tem %d caracteres\n", palavra, strlen (palavra));

  ⋮

  return 0;
}
```

- **void** `strcpy` (**char** `s1`[], **char** `s2`[]);

  copia o string armazenado em `s2` para o vetor `s1`.

  Exemplo:

```
#include <stdio.h>
#include <string.h>

int main () {
  ⋮

  char palavra[100]; /* declaração de um vetor de caracteres */
  char texto[200];  /* declaração de um vetor de caracteres */
  printf ("Entre com uma palavra: ");
  scanf ("%[^\n]", palavra);

  /* copia o string armazenado em palavra para o vetor texto */
  strcpy (texto, palavra);
  printf ("%s tem %d caracteres\n", texto, strlen (texto));

  ⋮

  return 0;
}
```

21.5 Exemplo de um Programa que usa String

O programa a seguir lê um string pelo teclado e imprime quantos caracteres tem o string lido.

```
1   # include <stdio.h>
2   # include <string.h>
3
4   int main () {
5     char nome [80]; /* string */
6     int  cont1, cont2;
7
8     printf ("Entre com um nome: ");
9     scanf (" %[^\n]", nome);
10
11    for (cont1=0; nome[cont1] != 0; cont1++);
12
13    cont2 = strlen (nome);
14
15    /* cont1 e cont2 sao iguais */
16
17    printf ("%s tem %d caracteres\n", nome, cont1);
18
19    return 0;
20
21  }
```

21.6 Problema 1

Faça um programa que leia uma frase e imprima esta frase usando apenas letras maiúculas.

```
# include <stdio.h>

int main () {
  char frase [80];
  int i;

  printf ("Entre com uma frase: ");
  scanf ("%[^\n]", frase);

  for (i=0; frase[i] != 0; i++) {
    if (frase[i] >= 'a' && frase[i] <= 'z')
    frase[i] = frase[i] - ('d' - 'D');
  }

  printf ("Frase Digitada em Maiusculas: %s\n", frase);

  return 0;
}
```

Na Linha 8, o usuário deve digitar uma frase seguida por um “enter”. Esta frase é armazenada no vetor `frase`.

Na Linha 10, o vetor `frase` é percorrido até encontrar o caractere ‘\0’ (código ASCII zero) que indica o fim de string. Para cada casa do vetor, verifica se é uma letra minúscula. Em caso afirmativo, transforma para maiúscula subtraindo da diferença entre uma letra minúscula e sua correspondente maiúscula (note que esta diferença é a mesma para qualquer letra - neste caso, foi escolhida a letra ‘d’).

21.7 Problema 2

Dados dois strings a e b, verifique quantas são as ocorrências do string b dentro de a.

Exemplo: se a é o string “Raras araras em Araraquara” e b é o string “ara”, o seu programa deve responder 5, pois o string “ara” aparece uma vez em “Raras”, duas em “araras” e outras duas em “Araraquara” (e não três já

que é feita a distinção entre maiúsculas e minúsculas).

Para resolver este problema, vamos fazer uma função que verifica se um string `menor` encaixa em um string `maior` a partir do índice `ind`.

```
int encaixa (char menor[], char maior[], int ind) {
  int i,j;
  j=ind;
  for (i=0; menor[i] != 0; i++) {
    if (menor[i] != maior[j]) {
      return 0;
    }
    j++;
  }
  return 1;
}
```

Agora é somente usar esta função para contar quantas vezes b encaixa em a.

```
int main () {
  char a[80], b[80];
  int  compr_a, compr_b, cont, i;

  printf ("Entre com uma frase a: ");
  scanf (" %[^\n]", a);

  printf ("Entre com uma palavra b: ");
  scanf (" %[^\n]", b);

  /* descobrindo o comprimento da frase a */
  for (compr_a=0; a[compr_a] != 0; compr_a++);

  /* descobrindo o comprimento da palavra b */
  for (compr_b=0; b[compr_b] != 0; compr_b++);


  for (i=cont=0; i < compr_a - compr_b + 1; i++) {
    cont = cont + encaixa (b, a, i);
  }

  printf ("%s aparece em %s %d vezes\n", b, a, cont);

  return 0;
}
```

22 Matrizes

Ronaldo F. Hashimoto e Carlos H. Morimoto

O objetivo desta aula é introduzir o tipo **matriz**. Ao final dessa aula você deverá saber:

- descrever o que são matrizes em C.
- Declarar matrizes.
- Como acessar elementos de uma matriz e percorrer uma matriz.
- Utilizar matrizes para resolver problemas computacionais.

22.1 Matrizes

Matrizes são estruturas indexadas (em forma matricial - como ilustrado na figura abaixo) utilizadas para armazenar dados de um mesmo tipo: **int**, **char**, **float** ou **double**.

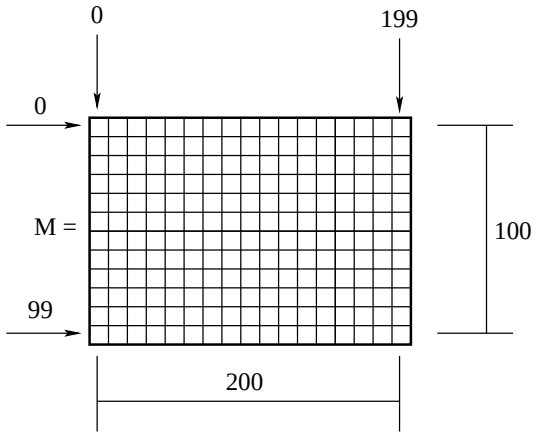


Figura 20: Uma matriz contem elementos de um mesmo tipo, com início em (0,0).

22.1.1 Declaração de Matrizes

A declaração de uma matriz é feita da seguinte forma:

`<tipo_da_matriz> <nome_da_matriz> [<numero_de_linhas>][<numero_de_colunas>];`

Exemplos:

- **int** `M[100][200];` ——— `100 é o número de linhas!` `200 é o número de colunas!`

  A declaração acima aloca uma matriz com 100 linhas e 200 colunas na memória. Cada casa da matriz guarda um **int**.

- **float** `x[20][30];` ——— `20 é o número de linhas!` `30 é o número de colunas!`

  A declaração acima aloca uma matriz com 20 linhas e 30 colunas na memória. Cada casa da matriz guarda um **float**.

Observação Importante:

1. Na **declaração de matriz**, o que está entre colchetes deve ser um **número constante**.
2. Assim, não é possível fazer algo deste tipo:

```
int nL = 20, nC = 30; /* num Linhas e num Colunas */
float x[nL][nC]; /* não se deve usar variáveis entre colchetes na declaração */
```

ou

```
int nL, nC;

printf ("Entre com nL>0 e nC>0: ");
scanf ("%d %d", &nL, &nC);

float x[nL][nC];
```

O correto seria:

```
int nL, nC;
float x[20][30]; /* o correto é declarar sempre tamanhos fixos */
```

22.1.2 Uso de Matrizes

- São usados índices para acessar uma linha e uma coluna de uma matriz.
- Os índices são números naturais.
- O índice da **primeira** linha é sempre **zero**.
- O índice da **primeira** coluna é sempre **zero**.

22.1.3 Exemplo de Uso de Matrizes

- Exemplo:

```
1       # include <stdio.h>
2
3       int main () {
4         int A[10][80], lin, col;
5
6         A[1][2] = 4; /* casa da linha 1 e coluna 2 recebe o inteiro 4 */
7         lin = 2; col = 3;
8         A[lin][col] = 5; /* casa de índice 2 do vetor v recebe o inteiro 3 */
9         A[A[lin-1][col-1] - 1][A[lin][col]] = 10; /* vc saberia dizer qual casa
10                                                  * da matriz A recebe o inteiro 10?
11                                                  */
12        return 0;
13      }
```

Na Linha 4, a matriz `A` com 10 linhas e 80 colunas é declarada:

	0	1	2	3	4	5		78	79
0	?	?	?	?	?	?	···	?	?
1	?	?	?	?	?	?	···	?	?
2	?	?	?	?	?	?	···	?	?
3	?	?	?	?	?	?	···	?	?
⋮	⋮	⋮	⋮	⋮	⋮	⋮	···	⋮	⋮
9	?	?	?	?	?	?	···	?	?

Na Linha 6, casa de linha 1 e coluna 2 da matriz `A` recebe o inteiro 4:

	0	1	2	3	4	5	...	78	79
0	?	?	?	?	?	?	...	?	?
1	?	?	4	?	?	?	...	?	?
2	?	?	?	?	?	?	...	?	?
3	?	?	?	?	?	?	...	?	?
⋮	⋮	⋮	⋮	⋮	⋮	⋮	...	⋮	⋮
9	?	?	?	?	?	?	...	?	?

Na Linha 8, casa de linha 2 e coluna 3 da matriz `A` recebe o inteiro 5:

	0	1	2	3	4	5	...	78	79
0	?	?	?	?	?	?	...	?	?
1	?	?	4	?	?	?	...	?	?
2	?	?	?	5	?	?	...	?	?
3	?	?	?	?	?	?	...	?	?
⋮	⋮	⋮	⋮	⋮	⋮	⋮	...	⋮	⋮
9	?	?	?	?	?	?	...	?	?

Na Linha 9, como `lin=2`, `col=3`, temos que `A[lin-1][col-1]=A[1][2]=4` e `A[lin][col]=A[2][3]=5`. Assim, temos que `A[A[lin-1][col-1] - 1][A[lin][col]]=A[4-1][5]=A[3][5]=10`. Dessa forma, no comando da Linha 9, a linha 3 e coluna 5 da matriz `A` recebe o inteiro 10:

	0	1	2	3	4	5	...	78	79
0	?	?	?	?	?	?	...	?	?
1	?	?	4	?	?	?	...	?	?
2	?	?	?	5	?	?	...	?	?
3	?	?	?	?	?	10	...	?	?
⋮	⋮	⋮	⋮	⋮	⋮	⋮	...	⋮	⋮
9	?	?	?	?	?	?	...	?	?

22.2 Percorrimento de Matrizes

Percorrer uma matriz significa visitar cada elemento da matriz (ou um subconjunto de elementos) de casa em casa em uma determinada ordem. Por exemplo, podemos percorrer apenas os elementos da diagonal principal de uma matriz quadrada, ou percorrer todos os elementos de uma matriz retangular, linha a linha, a partir da linha 0 (zero), e para cada linha, visitar os elementos de cada coluna, a partir da coluna 0 (zero). Nesse último caso é necessário saber o número de linhas e colunas que se deve fazer este percorrimento. Este número normalmente é guardado em duas variáveis inteiras (no nosso exemplo, as variáveis `nL` e `nC`).

Muitos problemas computacionais que envolvem matrizes têm como solução o uso de um padrão para percorrimento de matrizes.

Para os exemplos desta seção, vamos considerar a seguinte declaração de matriz:

```
int A[20][30];
```

e as variáveis inteiras

```
int lin, col, nL, nC, cont;
```

onde `nL` e `nC` são o número de linhas e colunas que devem ser consideradas na matriz `A`. É claro que neste caso, `nL` tem que ser menor que 20 e `nC` menor que 30.

22.2.1 Percorrimento de uma Linha:

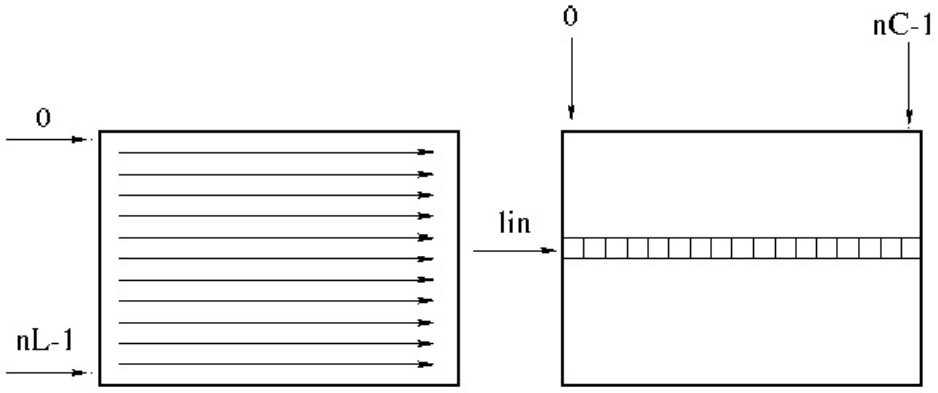


Figura 21: Percorrimento de uma linha de uma matriz.

Um padrão para percorrer uma linha `lin` da matriz `A` é usar um comando de repetição (no caso, vamos usar o comando **for**) com uma variável inteira `col` para o índice das colunas da matriz `A`:

```
/* para uma linha fixa lin */
for (col=0; col < nC; col++) {
  /* comandos usando a matriz A[lin][col] */
}
```

Exemplo:

```
cont = 0;
/* para uma linha fixa lin */
for (col=0; col < nC; col++) {
  A[lin][col] = cont;
  cont++;
}
```

22.2.2 Percorrimento Completo da Matriz:

Um padrão para percorrer completamente a matriz `A` (isto é, as `nL` linhas e as `nC` colunas) por linhas é usar dois comandos de repetição (no caso, vamos usar o comando **for**) com duas variáveis inteiras `lin` e `col`, um para percorrer as linhas e a outra para percorrer as colunas da matriz `A`:

```
for (lin=0; lin < nL; lin++) {
  for (col=0; col < nC; col++) {
    /* comandos usando a matriz A[lin][col] */
  }
}
```

O exemplo abaixo

```
for (lin=0; lin < nL; lin++) {
  for (col=0; col < nC; col++) {
    A[lin][col] = 0;
  }
}
```

inicializa as `nL` linhas e as `nC` colunas da matriz `A` com zero.

22.2.3 Observação sobre Percorrimento

Na declaração de uma matriz é definido um número fixo de linhas e colunas, uma vez que sempre deve-se colocar uma constante na definição do número de linhas e colunas da matriz. Por exemplo:

```
int A[20][30];
```

Mas, como podemos ver nos exemplos de percorrimento para ler e imprimir uma matriz, um usuário não necessariamente irá usar todas as linhas e colunas disponíveis da matriz. Note que no padrão de percorrimento por linhas deve sempre existir duas variáveis indicando quantas linhas e colunas da matriz estão sendo verdadeiramente usadas (variável `nL` e `nC` do padrão).

Assim, normalmente, em problemas computacionais que envolvem matrizes deve-se sempre ter duas variáveis inteiras associadas à matriz que diz quantas linhas e colunas da matriz estão sendo usadas (por exemplo, variável inteira `nL` e `nC` associadas à matriz `A` nos exemplos de leitura e impressão de matrizes).

22.3 Leitura de uma Matriz

Para leitura de uma matriz, devemos ler elemento a elemento usando o padrão de percorrimento por linhas.

```
1   # include <stdio.h>
2   # define MAX_L 100
3   # define MAX_C 200
4
5   int main () {
6     float A[MAX_L][MAX_C];
7     int lin, col, nL, nC;
8
9     printf ("Entre com 0<nL<%d: ", MAX_L);
10    scanf ("%d" &nL);
11
12    printf ("Entre com 0<nC<%d: ", MAX_C);
13    scanf ("%d" &nC);
14
15    /* percorrer a matriz A elemento a elemento
16     * colocando o valor lido pelo teclado */
17    for (lin=0; lin < nL; lin++) {
18      for (col=0; col < nC; col++) {
19        printf ("Entre com A[%d][%d] = ", lin, col);
20        scanf ("%f", &A[lin][col]);
21      }
22    }
23
24    return 0;
25  }
```

Observe com cuidado a linha do programa utilizada para ler o elemento da linha `lin` e coluna `col` da matriz `A`:

```
scanf ("%f", &A[lin][col]);
```

A linha `lin` e a coluna `col` da matriz `A`, ou seja, `A[lin][col]`, é utilizada da mesma forma que utizamos qualquer variável até o momento, ou seja, precedida pelo caractere '&'.

Note que neste exemplo definimos as constantes `MAX_L` e `MAX_C` usando o "comando" `define`. Observe que `MAX_L` e `MAX_C` são constantes e não variáveis. Para saber mais sobre a definição de constantes, veja o material didático **Alguns Detalhes da Linguagem C**.

22.4 Impressão de uma Matriz

Para impressão de uma matriz, devemos imprimir elemento a elemento usando o padrão de percorrimento por linhas.

```c
#include <stdio.h>

#define MAX_L 100
#define MAX_C 200

int main () {
  float A[MAX_L][MAX_C];
  int lin, col, nL, nC;

  printf ("Entre com 0<nL<%d: ", MAX_L);
  scanf ("%d" &nL);

  printf ("Entre com 0<nC<%d: ", MAX_C);
  scanf ("%d" &nC);

  /* percorrer a matriz A elemento a elemento
   * imprimindo o valor de cada casa */
  for (lin=0; lin<nL; lin++) {
    for (col=0; col<nC; col++) {
      printf ("%f ", A[lin][col]);
    }
    printf ("\n");
  }

  return 0;
}
```

22.5 Exercícios Comentados

22.5.1 Exercício 1

Faça um programa que leia um inteiro $n < 100$ e os elementos de uma matriz real quadrada $A_{n \times n}$ e verifica se a matriz A tem uma linha, coluna ou diagonal composta apenas por zeros.

Percorrimento de uma Linha de uma Matriz:

Para verificar se uma matriz `A` tem uma linha com todos elementos nulos, devemos percorrer uma linha `lin` da matriz `A`.

Ora, nós já conhecemos o padrão para percorrer uma linha `lin` da matriz `A`:

```c
/* para uma linha fixa lin */
for (col=0; col<n; col++) {
  /* comandos usando a matriz A[lin][col] */
}
```

Para contar quantos elementos nulos tem uma linha, podemos usar o padrão de percorrimento de uma linha da seguinte maneira:

```
cont = 0;
/* para uma linha fixa lin */
for (col=0; col<n; col++) {
  if (A[lin][col] == 0)
    cont++;
}
```

Neste exemplo, o padrão conta quantos elementos nulos tem a linha `lin`. Se quisermos saber se a linha `lin` tem todos os elementos nulos, basta comparar se `cont` é igual a `n`. Assim:

```
cont = 0;
/* para uma coluna fixa col */
for (lin=0; lin<n; lin++) {
  if (A[lin][col] == 0)
    cont++;
}
if (cont == n)
  printf ("A linha %d tem todos elementos nulos\n", i);
```

Assim, para verificar se uma matriz `A` tem uma linha com todos elementos nulos, devemos verificar cada linha `lin` da matriz:

```
linha_nula = 0;
for (lin=0; lin<n; lin++) {
  cont = 0;
  /* para uma linha lin */
  for (col=0; col<n; col++) {
    if (A[lin][col] == 0)
      cont++;
  }
  if (cont == n)
    linha_nula = 1;
}
if (linha_nula == 1)
  printf ("Matriz tem uma linha com todos elementos nulos\n",);
```

Percorrimento de uma Coluna de uma Matriz:

Para verificar se uma matriz `A` tem uma coluna com todos elementos nulos, devemos saber como percorrer uma coluna `col` da matriz `A`.

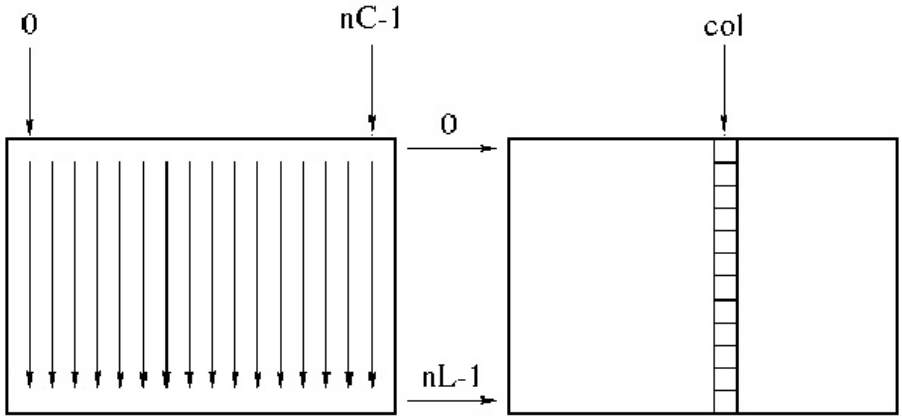


Figura 22: Percorrimento de uma coluna de uma matriz.

Um padrão para percorrer uma coluna `col` de uma matriz `A` é usar um comando de repetição (no caso, vamos

usar o comando **for**) com uma variável inteira `lin` para o índice das linhas da matriz `A`:

```
/* para uma coluna fixa col */
for (lin=0; lin<n; lin++) {
  /* comandos usando a matriz A[lin][col] */
}
```

Exemplos:

```
cont = 0;
/* para uma coluna fixa col */
for (lin=0; lin<n; lin++) {
  A[lin][col] = cont;
  cont++;
}

cont = 0;
/* para uma coluna fixa col */
for (lin=0; lin<n; lin++) {
  if (A[lin][col] == 0)
    cont++;
}
```

No último exemplo, o padrão conta quantos elementos nulos tem a coluna `col`. Se quisermos saber se a coluna `col` de uma matriz $A_{n \times n}$ tem todos os elementos nulos, basta comparar se `cont` é igual a `n`. Assim:

```
cont = 0;
/* para uma coluna fixa col */
for (lin=0; lin<n; lin++) {
  if (A[lin][col] == 0)
    cont++;
}
if (cont == n)
  printf ("A coluna %d tem todos elementos nulos\n", j);
```

Assim, para verificar se uma matriz quadrada `A` tem uma coluna com todos elementos nulos, devemos verificar cada coluna `col` da matriz:

```
coluna_nula = 0;
for (col=0; col<n; col++) {
  cont = 0;
  /* para uma coluna col */
  for (lin=0; lin<n; lin++) {
    if (A[lin][col] == 0)
      cont++;
  }
  if (cont == n)
    coluna_nula = 1;
}
if (coluna_nula == 1)
  printf ("Matriz tem uma coluna com todos elementos nulos\n",);
```

Percorrimento da Diagonal Principal de uma Matriz:

Para verificar se uma matriz quadrada $A_{n \times n}$ tem a diagonal principal com todos elementos nulos, devemos saber como percorrer esta diagonal da matriz `A`.

Como na diagonal principal temos que a linha é igual a coluna, um padrão para percorrer a diagonal principal de `A` é usar um comando de repetição (no caso, vamos usar o comando **for**) com uma variável inteira `lin` para o índice das linhas e colunas da matriz `A`:

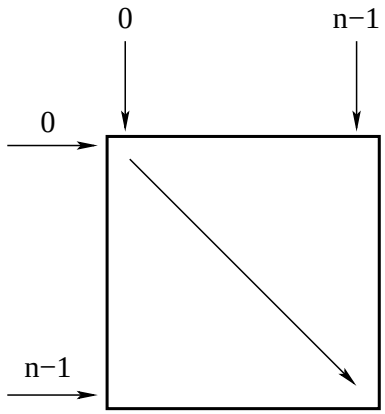


Figura 23: Percorrimento da diagonal.

```
for (lin=0; lin<n; lin++) {
  /* comandos usando a matriz A[lin][lin] */
}
```

Exemplos:

```
cont = 0;
for (lin=0; lin<n; lin++) {
  A[lin][lin] = cont;
  cont++;
}

cont = 0;
for (lin=0; lin<n; lin++) {
  if (A[lin][lin] == 0)
    cont++;
}
```

No último exemplo, o padrão conta quantos elementos nulos tem a diagonal principal. Se quisermos saber se a diagonal principal de uma matriz $A_{n\times n}$ tem todos os elementos nulos, basta comparar se `cont` é igual a `n`. Assim:

```
cont = 0;
for (lin=0; lin<n; lin++) {
  if (A[lin][lin] == 0)
    cont++;
}
if (cont == n)
  printf ("A diagonal principal tem todos elementos nulos\n");
```

Percorrimento da Diagonal Secundária de uma Matriz:

Para verificar se uma matriz quadrada $A_{n\times n}$ tem a diagonal secundária com todos elementos nulos, devemos saber como percorrer esta diagonal da matriz `A`.

Como na diagonal secundária temos que a soma da linha com a coluna é igual a `n−1` (ou seja, para uma linha `lin`, a coluna deve ser `n−1−i`), um padrão para percorrer a diagonal secundária de `A` é usar um comando de repetição (no caso, vamos usar o comando **for**) com uma variável inteira `lin` para o índice das linhas e colunas da matriz `A`:

```
for (lin=0; lin<n; lin++) {
  /* comandos usando a matriz A[lin][n-1-lin] */
}
```

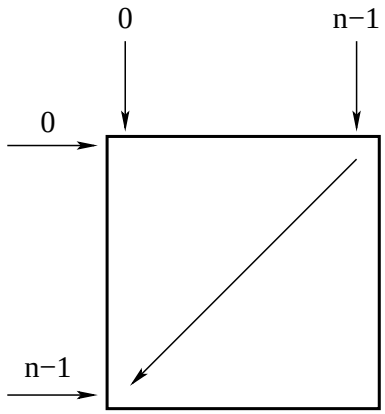


Figura 24: Percorrimento da diagonal secundária.

Exemplos:

```
cont = 0;
for (lin=0; lin<n; lin++) {
  A[lin][n-1-lin] = cont;
  cont++;
}

cont = 0;
for (lin=0; lin<n; lin++) {
  if (A[lin][n-1-lin] == 0)
    cont++;
}
```

No último exemplo, o padrão conta quantos elementos nulos tem a diagonal secundária. Se quisermos saber se a diagonal secundária de uma matriz $A_{n\times n}$ tem todos os elementos nulos, basta comparar se `cont` é igual a `n`. Assim:

```
cont = 0;
for (lin=0; lin<n; lin++) {
  if (A[lin][n-1-lin] == 0)
    cont++;
}
if (cont == n)
  printf ("A diagonal secundária tem todos elementos nulos\n");
```

Juntando Tudo

Fica como exercício você fazer um programa que resolva o Exercício 1, ou seja, fazer um programa que leia um inteiro $n < 100$ e os elementos de uma matriz real $A_{n\times n}$ e verifica se a matriz A tem uma linha, coluna ou diagonal composta apenas por zeros.

22.5.2 Exercício 2

Dado $0 < n < 200$ e uma matriz real $A_{n\times n}$, verificar se A é simétrica.

Uma matriz $A_{n\times n}$ é simétrica se, e somente se, A é igual a sua transposta, ou seja, $A = A^t$.

Neste caso, temos que verificar se cada $A[lin][col]$ é igual a $A[col][lin]$ como indicado na figura. Note que devemos percorrer somente uma parte da matriz, no caso, a parte superior da matriz.

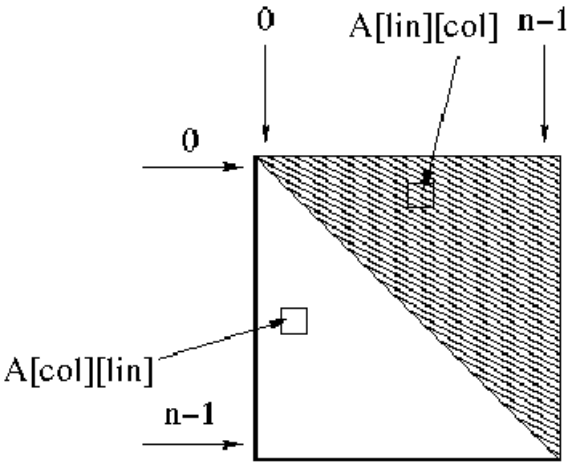


Figura 25: Matriz simétrica.

Percorrimento da Parte Superior de Matrizes por Linha

Para uma linha `lin`, temos que começar a percorrer as colunas a partir da coluna `lin+1` até a última coluna `n−1`, como mostra a figura 26.

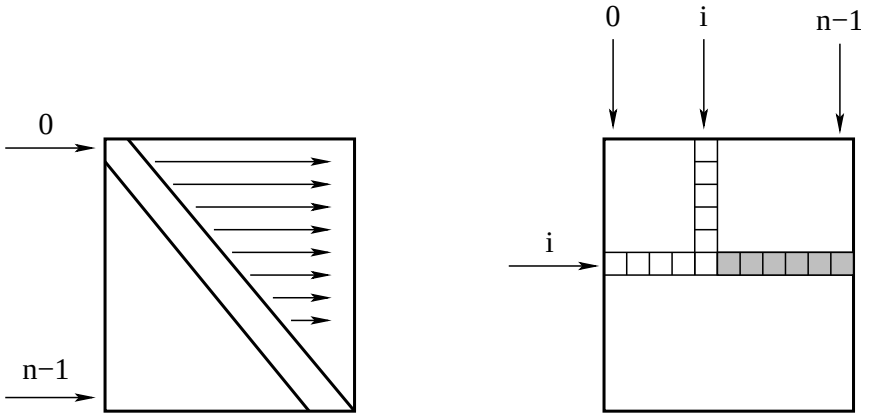


Figura 26: Percorrimento da parte superior da matriz por linha.

Assim, um padrão para percorrer uma linha `lin` da parte superior de uma matriz `A` é usar um comando de repetição (no caso, vamos usar o comando **for**) com uma variável inteira `col` para o índice das colunas da matriz `A`:

```
/* para uma linha fixa lin */
for (col=lin+1; col<n; col++) {
  /* comandos que fazem algo com A[lin][col] */
}
```

Exemplo:

```
cont = 0;
/* para uma linha fixa lin */
for (col=lin+1; col<n; col++) {
  A[lin][col] = cont;
  cont++;
}
```

Um padrão para percorrer completamente a parte superior da matriz `A` por linhas é usar dois comandos de repetição (no caso, vamos usar o comando **for**) com duas variáveis inteiras `lin` e `col`, um para percorrer as linhas e a outra para percorrer as colunas da matriz `A`:

```
for (lin=0; lin<n; lin++) {
  /* para uma linha fixa lin */
  for (col=lin+1; col<n; col++) {
    /* comandos que fazem algo com A[lin][col] */
  }
}
```

Exemplo:

```
cont = 0;
for (lin=0; lin<n; lin++) {
  /* para uma linha fixa lin */
  for (col=lin+1; col<n; col++) {
    A[lin][col] = cont;
    cont++;
  }
}
```

Assim, para verificar se uma matriz real $A_{n \times n}$, verificar se A é simétrica, podemos fazer:

```
simetrica = 1;
for (lin=0; lin<n; lin++) {
  /* para uma linha fixa lin */
  for (col=lin+1; col<n; col++) {
    if (A[lin][col] != A[col][lin])
      simetrica = 0;
  }
}
printf ("Matriz ");
if (simetrica == 0) {
  printf ("nao ");
}
printf ("eh Simetrica\n");
```

Solução Completa:

```c
# include <stdio.h>

# define MAX 100

int main () {
  int lin, col, n, simetrica = 1;
  float A[MAX][MAX];

  printf ("Entre com 0<n<100: ");
  scanf ("%d" &n);

  /* percorrer a matriz A elemento a elemento
  * colocando o valor lido pelo teclado */
  for (lin=0; lin<n; lin++) {
    /* para uma linha fixa lin */
    for (col=0; col<n; col++) {
      printf ("Entre com A[%d][%d] = ", lin, col);
      scanf ("%f", &A[lin][col]);
    }
  }

  /* verificando se eh simetrica */
  for (lin=0; lin<n; lin++) {
    /* para uma linha fixa i */
    for (col=lin+1; col<n; col++) {
      if (A[lin][col] != A[col][lin])
      simetrica = 0;
    }
  }

  /* Impressao da Resposta Final */
  printf ("Matriz ");
  if (simetrica == 0) {
    printf ("nao ");
  }
  printf ("eh Simetrica\n");
  return 0;
}
```

22.6 Erros Comuns

Ao desenvolver seus programas com matrizes, preste atenção com relação aos seguintes detalhes:

- **índices inválidos:** tome muito cuidado, especialmente dentro de um **while** ou **for**, de não utilizar índices negativos ou maiores que o tamanho máximo designado para as linhas e colunas da matriz.
- A **definição do tamanho das linhas e colunas da matriz** se faz na declaração da matriz. Os tamanhos das linhas e colunas são constantes; só mudando a sua declaração é que podemos alterar estes tamanhos. Isso significa que podemos estar "desperdiçando" algum espaço da memória por não estar usando todas as casas da matriz. Não cometa o erro de ler `nL` e `nC`, onde `nL` e `nC` seriam os tamanhos das linhas e colunas da matriz, e tentar "declarar" a matriz em seguida.

22.7 Percorrimento de Matrizes

Muitos problemas computacionais que envolvem matrizes têm como soluções o uso de um padrão para percorrimento de matrizes. Nesta aula aprendemos:

- Percorrimento de uma Linha de uma Matriz.
- Percorrimento Completo da Matriz por Linhas.
- Percorrimento de uma Coluna de uma Matriz.
- Percorrimento Completo da Matriz por Colunas.
- Percorrimento da Diagonal Principal de uma Matriz.
- Percorrimento da Diagonal Secundária de uma Matriz.
- Percorrimento da Parte Superior de Matrizes por Linha.

Há muitas outras forma de percorrer matrizes, como mostra a figura 27.

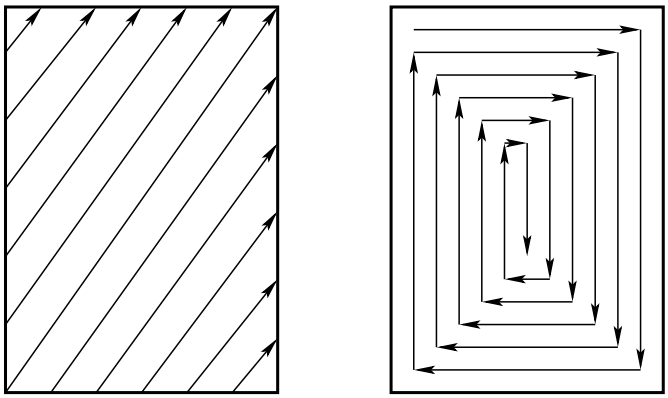


Figura 27: Outras formas de percorrer uma matriz.

22.8 Exercícios Recomendados

1. Escreva um programa que, dadas duas matrizes $A_{m\times n}$ e $B_{n\times p}$, calcula a matriz $C_{m\times p}$ que é o produto de A por B. Note que, para ler as matrizes, é necessário primeiro ler os seus tamanhos $m, n,$ e p.
2. Imprimir as n primeiras linhas do triângulo de Pascal.
3. Um jogo de palavras cruzadas pode ser representado por uma matriz $A_{m\times n}$ onde cada posição da matriz corresponde a um quadrado do jogo, sendo que 0 (zero) indica um quadrado branco e -1 indica um quadrado preto. Indicar na matriz as posições que são início de palavras horizontais e/ou verticais nos quadrados correspondentes (substituindo os zeros), considerando que uma palavra deve ter pelo menos duas letras. Para isso, numere consecutivamente tais posições.

   Exemplo: Dada a matriz:

$$\begin{pmatrix} 0 & -1 & 0 & -1 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & -1 & -1 \end{pmatrix}$$

   A saída deverá ser:

$$\begin{pmatrix} 1 & -1 & 2 & -1 & -1 & 3 & -1 & 4 \\ 5 & 6 & 0 & 0 & -1 & 7 & 0 & 0 \\ 8 & 0 & -1 & -1 & 9 & 0 & -1 & 0 \\ -1 & 10 & 0 & 11 & 0 & -1 & 12 & 0 \\ 13 & 0 & -1 & 14 & 0 & 0 & -1 & -1 \end{pmatrix}$$

23 Matrizes, Ponteiros e Funções

Ronaldo F. Hashimoto e Carlos H. Morimoto

O objetivo desta aula é relacionar o tipo **matrizes** com ponteiros e assim entender como utilizar matrizes como parâmetros de funções. Ao final dessa aula você deverá saber:

- Descrever como matrizes são armazenadas na memória.
- Descrever a relação entre matrizes e ponteiros.
- Utilizar matrizes como parâmetros de funções.

23.1 Matrizes

Vimos na aula anterior que **matrizes** são estruturas indexadas (em forma matricial) utilizadas para armazenar dados de um mesmo tipo: **int**, **char**, **float** ou **double**. Por exemplo, a declaração

```
int M[100][200]; /* declara uma matriz de inteiros
                  * de nome M com 100 linhas e 200 colunas
                  */
```

alocaria uma estrutura de dados da forma:

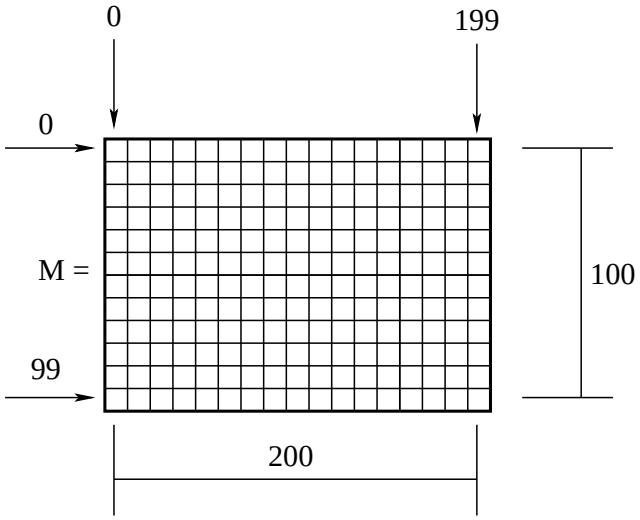


Figura 28: Estrutura de uma matriz int M[100][200].

Uma pergunta que poderíamos fazer é como uma matriz fica armazenada na memória. Como a memória do computador é linear, o armazenamento dos elementos é feito colocando-se cada linha da matriz uma em seguida da outra. Assim, por exemplo, os elementos em uma matriz declarada como `int M[100][200]` são armazenados na memória do computador conforme mostra a figura 29.

Isso significa que para a matriz `M`, os seus elementos são armazenados na memória da seguinte maneira: os 200 elementos da primeira linha `M[0][0],M[0][1],...,M[0][199]`, seguidos pelos elementos da segunda linha `M[1][0],M[1][1],...,M[1][199]` e assim por diante até a última linha `M[99][0],M[99][1],...,M[99][199]`. Dessa forma, a disposição dos 20.000 elementos da matriz `M` na memória seria:

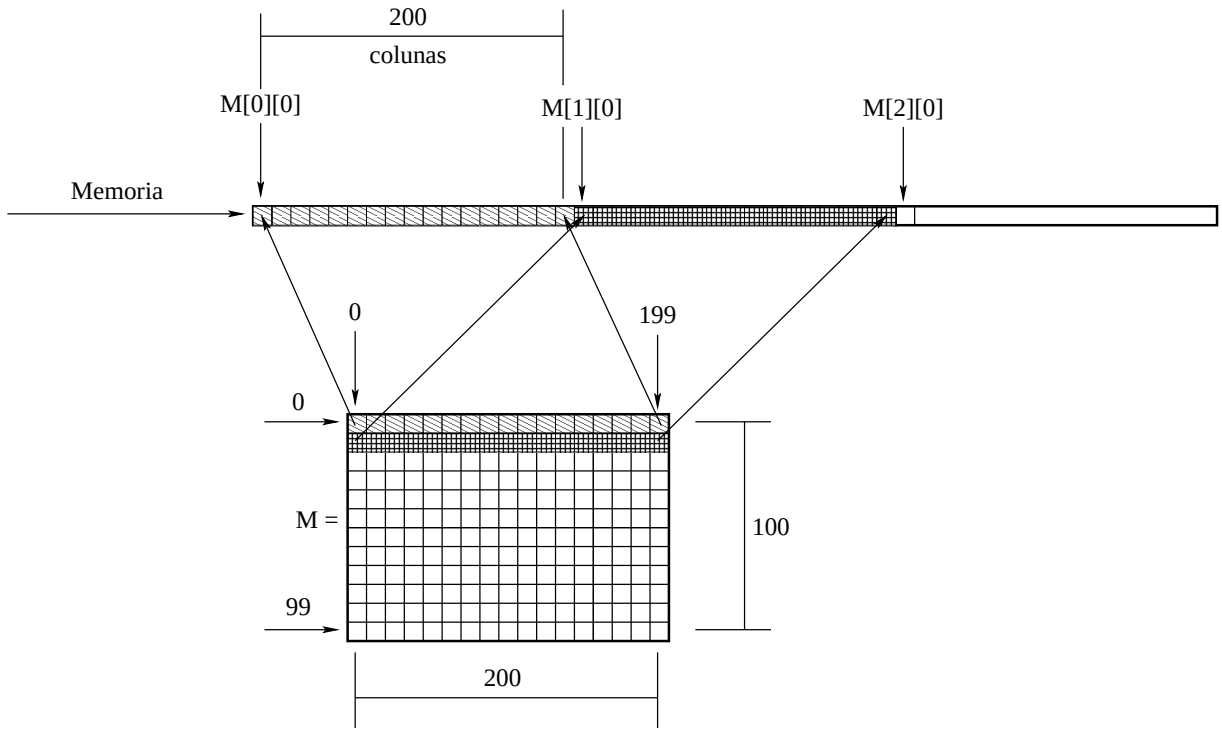


Figura 29: Estrutura da matriz na memória do computador.

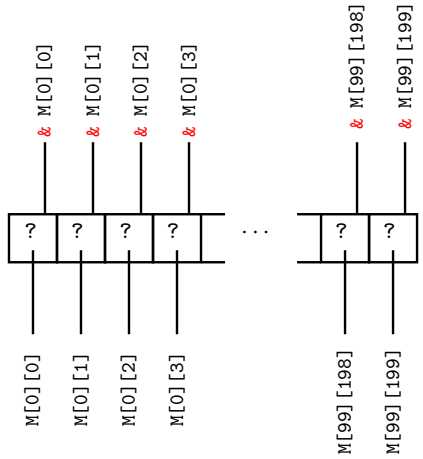


Figura 30: Disposição dos 20.000 elementos da matriz `M` na memória.

Observando a figura 30, podemos notar que cada elemento da matriz `M` tem um endereço associado. Para efeitos **didáticos**, vamos supor que o endereço de `M[0][0]` é um número inteiro, digamos 10, ou seja, `&M[0][0]`→10 e que os endereços das casas seguintes são números inteiros consecutivos a partir do endereço 10. Assim, temos, para efeitos **didáticos**, que `&M[0][0]`→10, `&M[0][1]`→11, `&M[0][2]`→12, ..., `&M[99][198]`→20.008, `&M[99][199]`→20.009. Com isto, é possível saber o endereço de qualquer casa de `M` conhecendo-se o endereço de `M[0][0]`. Por exemplo, endereço de `M[0][78]` é `&M[0][0]`+78=10+78=88. Agora, para ver se você entendeu os conceitos até aqui, vamos fazer uma pergunta para você.

Você saberia me dizer qual é o endereço de `M[78][21]`?[6]

Para identificar o endereço de memória associado a um determinado elemento `M[i][j]`, é feita internamente uma conta de endereçamento: o endereço do elemento `M[i][j]` é `&M[0][0] + i · nC + j`, onde `nC` é o número de colunas da matriz. Este é um detalhe interno, que é feito automaticamente pelo compilador. Nos seus programas, você

[6] `&M[0][0] + i * nC + j` = 10 + (78 * 200 + 21), onde `nC` é o número de colunas da matriz.

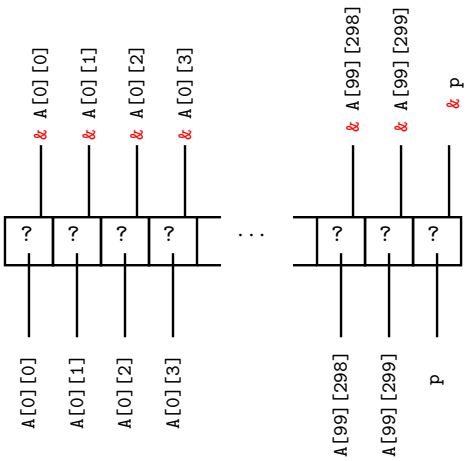


Figura 31: Possível configuração da memória para as variáveis A e p.

apenas precisa se preocupar em acessar os elementos escrevendo `M[i][j]`. O importante é observarmos que o compilador necessita saber o número `nC` de colunas da matriz, no nosso exemplo `nC` = 200, para fazer a conta de endereçamento. Esta informação nos será útil adiante.

23.1.1 Exercício

```
int N[200][100]; /* declara uma matriz de inteiros
                  * de nome N com 200 linhas e 100 colunas
                  */
```

Suponha que o endereço de `N[0][0]` é um número inteiro, digamos 10, ou seja, `&N[0][0]`→10 e que os endereços das casas seguintes são números inteiros consecutivos a partir do endereço de 10. Assim, temos que `&N[0][0]`→10, `&N[0][1]`→11, `&N[0][2]`→12, . . ., `&N[99][198]`→20.008, `&N[99][199]`→20.009.

Você saberia me dizer qual é o endereço de `N[78][21]`?[7] Por que é diferente do endereço de `M[78][21]`?

23.1.2 Observação

Na linguagem C não existe verificação de índices fora da matriz. Quem deve controlar o uso correto dos índices é o programador. Além disso, o acesso utilizando um índice errado pode ocasionar o acesso de outra variável na memória. Se o acesso à memória é indevido você recebe a mensagem *"segmentation fault"*.

23.2 Matrizes e Ponteiros

Considere agora o seguinte trecho de código:

```
int A[100][300];
int *p; /* ponteiro para inteiro */
```

A figura 31 ilustra uma possível configuração da memória para as variáveis A e p.

Podemos utilizar a sintaxe normal para fazer um ponteiro apontar para uma casa da matriz:

```
p = &A[0][0]; /* p aponta para a A[0][0] */
```

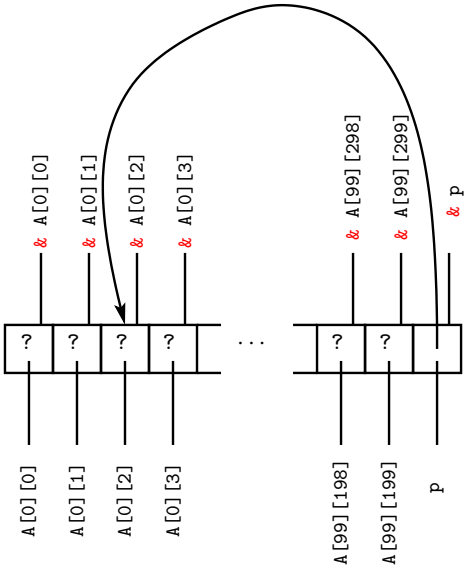


Figura 32: Ponteiro apontando para a matriz (elemento A[0][0]).

O ponteiro `p` recebe o endereço de `A[0][0]`, ou seja, `p` aponta para `A[0][0]`.

Na linguagem C, uma matriz pode ser considerada um vetor de vetores, o que traz várias vantagens aos programadores dessa linguagem.

- Uma das conveniências é utilizar o nome de uma matriz com apenas o primeiro índice para referenciar uma linha em particular, como mostra pedaço de código abaixo:

  ```
  p = A[1];
  ```

  Dessa forma `p` pode ser utilizado para acessar os elementos da linha 1 da matriz `A`. Dada a forma como as matrizes são armazenadas na memória, não é possível referenciar diretamente a uma coluna da matriz usando apenas um índice, ou seja, algo como:

  ```
  p = A[][1]; /* atribuição inválida. */
  ```

  Se for necessário utilizar uma coluna da matriz como vetor, você deve copiar os elementos da coluna para um vetor auxiliar.

- Lembre-se que ponteiros podem ser utilizados para acessar elementos de um vetor usando a mesma sintaxe de vetores (`nome_do_vetor[índice]`) com o ponteiro. Podemos utilizar essa mesma sintaxe também com matrizes. Assim se fizermos:

  ```
  p = A[0];
  ```

  podemos acessar o elemento que está na linha `i` e coluna `j` de `A` fazendo `p[i*300+j]` (300 é o número de colunas da matriz `A`), ou seja, ambos `p[i*300+j]` e `A[i][j]` acessam a casa da linha `i` e coluna `j` da matriz `A`. Exemplo:

  ```
  int A[100][300], i, j;
  int *p; /* ponteiro para inteiro */
  i = 3; j = 4;
  p = A[0]; /* p aponta para o 1o elemento da 1a linha de A */
            /* Equivale a fazer p = &A[0][0] */
  p[i*300+j] = 4; /* equivale a fazer M[i][j] = 4 */
  ```

[7] 10 + (78 * 100 + 21)

Mas se fazemos:

```
p = &A[3][0]; /* Equivale a p = A[3] */
```

então, `p[0]` é o elemento `A[3][0]`, `p[1]` é o elemento `A[3][1]`, `p[2]` é o elemento `A[3][2]`, e assim por diante.

Observe que a construção `p[i][j]` **não** é válida por possuir 2 pares de colchetes. Portanto a atribuição `p = A;` não é correta pois a linguagem C permite que um ponteiro seja utilizado apenas com uma dimensão (ou seja, apenas um par de colchetes), e uma matriz possui 2 dimensões. Uma matriz também não pode ser considerada um ponteiro para ponteiro, também devido a sua estrutura particular onde cada linha é seguida imediatamente por outra. Assim, a atribuição no seguinte pedaço de código:

```
int **p;
int A[100][100];
p = A;
```

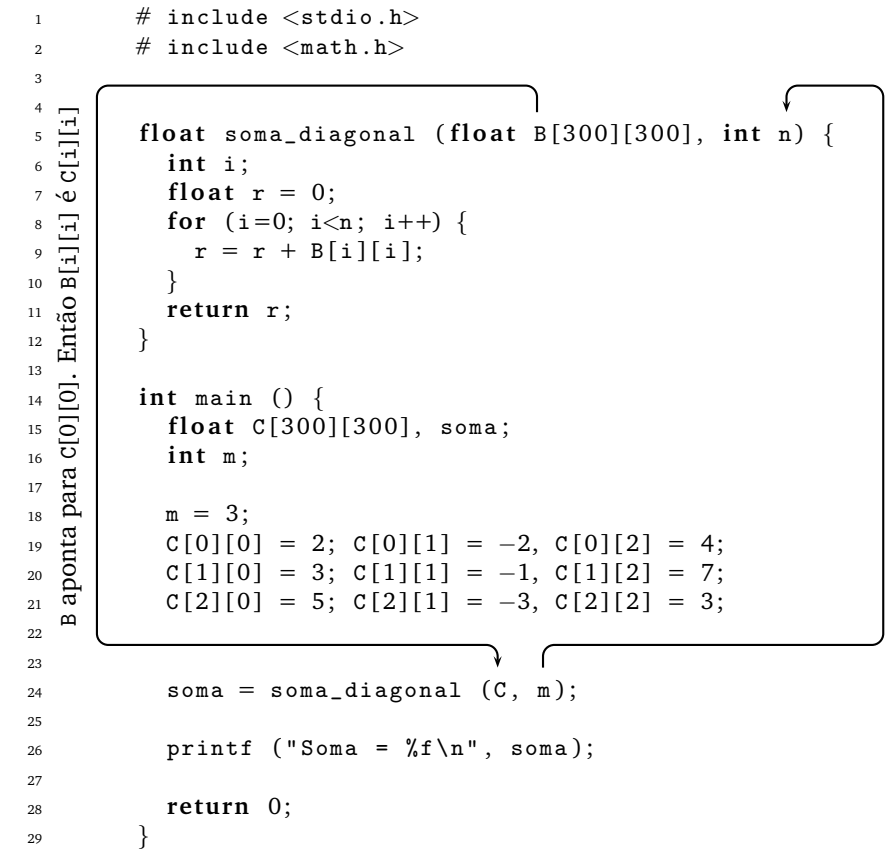
também não é válida.

23.3 Matrizes como Parâmetro de Funções

A forma de declaração de uma matriz como parâmetro de função é a mesma que vimos para declarar matrizes, ou seja, indicando o nome da matriz, e entre colchetes o número de linhas e o número de colunas. Exemplo:

```
# include <math.h>
float soma_diagonal (float B[300][300], int n) {
  int i;
  float r = 0;
  for (i=0; i<n; i++) {
    r = r + B[i][i];
  }
  return r;
}
```

Esta função recebe uma matriz de reais `B` de tamanho 300×300 das quais somente `n` linhas e colunas estão sendo consideradas e devolve a soma dos elementos de sua diagonal.



O parâmetro `B` da função `soma_diagonal` aponta para a variável `C[0][0]` da função `main`. Então `B[i][i]` na função `soma_diagonal` é exatamente `C[i][i]` da função `main`.

23.4 Exemplo de Função com Matriz como Parâmetro

O nome de uma matriz dentro de parâmetro de uma função é utilizado como sendo um ponteiro para o primeiro elemento da matriz que está sendo usada na hora de chamar a função.

Exemplo de declaração de funções com matrizes como parâmetros:

```
1       # define MAX 200
2
3
4       float f (float M[MAX][MAX]) {
5         float s;
6         /* declaração da função f */
7         ...
8         M[i][j] = 4;
9         ...
10        return s;
11      }
12
13      int main () {
14        float a, A[MAX][MAX]; /* declaração da variável a e da matriz A */
15        ...
16        /* outras coisas do programa */
17
18
19        a = f (A);   /* observe que a matriz é passada apenas pelo nome */
20
21        ...
22
23        return 0;
24      }
```

M aponta para A[0][0].

Na Linha 19, a chamada da função `f` faz com que o ponteiro `M` receba `&A[0][0]`, ou seja, faz com que o ponteiro `M` aponte para `A[0][0]`.

Na Linha 8, temos o comando `M[i][j] = 4`. Como `M` está apontando para `A[0][0]`, então `M[i][j] = 4` é o mesmo que fazer `A[i][j] = 4`. Assim, na Linha 8, dentro da função `f`, estamos mudando o conteúdo da casa de linha `i` e coluna `j` da matriz `A` da função `main`.

23.4.1 Exemplo

```
1     # include <stdio.h>
2
3     # define MAX 100
4
5     int f (int M[MAX][MAX], int n) {
6       n = 10;
7       M[2][3] = 4;
8       return 0;
9     }
10
11    int main () {
12      int A[MAX][MAX], m, a;
13      m = 15;
14      A[2][3] = 5;
15      a = f (A, m);
16      return 0;
17    }
```

Na Linha 14, temos o comando `A[2][3] = 5`, ou seja `A[2][3]` contém o valor `5`.

Na Linha 15, a chamada da função `f` faz com que o ponteiro `M` receba `&A[0][0]`, ou seja, faz com que o ponteiro `M` aponte para `A[0][0]`.

Na Linha 7, temos o comando `M[2][3] = 4`. Como `M` está apontando para `A[0][0]`, então `M[2][3] = 4` é o mesmo que fazer `A[2][3] = 4`. Assim, na Linha 7, dentro da função `f`, estamos mudando o conteúdo da casa de linha 2

e coluna 3 da matriz `A` da função `main`. Dessa forma, depois da execução da função `f`, `A[2][3]` conterá o valor 4 e não 5 como foi inicializado.

23.5 Problema

(a) Faça uma função que recebe como entrada um inteiro n, uma matriz inteira $A_{n \times n}$ e devolve três inteiros: k, Lin e Col. O inteiro k é um maior elemento de A e é igual a $A[Lin][Col]$.

Obs.: Se o elemento máximo ocorrer mais de uma vez, indique em Lin e Col qualquer uma das possíveis posições.

Exemplo: se $A = \begin{pmatrix} 3 & 7 & 1 \\ 1 & 2 & 8 \\ 5 & 3 & 4 \end{pmatrix}$ então $\begin{cases} k = 8 \\ Lin = 1 \\ Col = 2 \end{cases}$

Solução: vamos chamar esta função de `maior`. A função `maior` deve receber uma matriz A (vamos supor que a matriz não vai ter mais que 100 linhas e colunas) e um inteiro n indicando (das 100 linhas e colunas reservadas) quantas linhas e colunas de fato estão sendo utilizadas. Como esta função deve devolver três valores (k, Lin e Col), então vamos usar três ponteiros. Dessa forma, o protótipo da função `maior` é

```
# define MAX 100

void maior (int A[MAX][MAX], int n, int *k, int *Lin, int *Col);
```

Esta função recebe uma matriz `A` e um inteiro `n` e devolve três valores `*k`, `*Lin` e `*Col`, via ponteiros. Para isto, vamos percorrer a matriz `A` (usando o padrão de percorrimento de matriz por linha) e verificando se cada elemento `A[i][j]` da matriz é maior que `*k`. Em caso afirmativo, `*k`, `*Lin` e `*Col` são atualizados para `A[i][j]`, `i` e `j`, respectivamente.

```
# define MAX 100

void maior (int A[MAX][MAX], int n, int *k, int *Lin, int *Col) {
  int i, j;

  /* percorrimento da matriz A por linha */
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      if (A[i][j] > *k) {
        *k = A[i][j];
        *Lin = i;
        *Col = j;
      }
    }
  }

}
```

Se você observou bem a solução acima, faltou inicializar `*k`, `*Lin` e `*Col`. Para esta função encontrar o maior corretamente, `*k` deve ser inicializado com qualquer elemento da matriz `A`. Assim, vamos inicializar `*k` com o primeiro elemento da matriz. Assim , `*k`, `*Lin` e `*Col` são inicializados com `A[0][0]`, 0 e 0, respectivamente.

```
# define MAX 100

void maior (int A[MAX][MAX], int n, int *k, int *Lin, int *Col) {
  int i, j;

  *k = A[0][0];
  *Lin = *Col = 0;

  /* percorrimento da matriz A por linha */
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      if (A[i][j] > *k) {
        *k = A[i][j];
        *Lin = i;
        *Col = j;
      }
    }
  }

}
```

(b) Faça um programa que, dado um inteiro n e uma matriz quadrada de ordem n, cujos elementos são todos **inteiros positivos**, imprime uma tabela onde os elementos são listados em ordem decrescente, acompanhados da indicação de linha e coluna a que pertencem. Havendo repetições de elementos na matriz, a ordem é irrelevante. Utilize obrigatoriamente a função da parte (a), mesmo que você não a tenha feito.

Exemplo: No caso da matriz $A = \begin{pmatrix} 3 & 7 & 1 \\ 1 & 2 & 8 \\ 5 & 3 & 4 \end{pmatrix}$, a saída poderia ser:

Elem	Linha	Coluna
8	1	2
7	0	1
5	2	0
4	2	2
3	0	0
3	2	1
2	1	1
1	0	2
1	1	0

Como a matriz A somente tem valores **inteiros positivos**, uma estratégia para solucionar este problema é usar a função `maior` que devolve o maior elemento da matriz e sua respectiva posição e colocar nesta posição um inteiro negativo, digamos -1. Chama a função `maior` novamente e devolve o maior elemento da matriz e sua respectiva posição (não vai devolver o primeiro maior, pois esta posição foi marcada com -1); colocar nesta posição -1 e chamar a função `maior` novamente; até que todos os elementos da matriz sejam negativos (quando a função `maior` devolver o número -1). Vejamos então um trecho que programa que faz esta parte:

```
printf ("Elem  Linha  Coluna\n");
k = 1; /* qualquer valor positivo */
while (k != -1) {
  maior (A, n, &k, &L, &C);
  if (k != -1) {
    printf ("%d %d %d\n", k, L, C);
    A[L][C] = -1;
  }
}
```

Fica como exercício você colocar este trecho acima em um programa completo.

24 Estruturas Heterogêneas (struct)

Routo Terada

As estruturas que conhecemos como matriz ou vetor são homogêneas, isto é, todos os elementos são do mesmo tipo. Por exemplo, todos são do tipo **int** ou todos do tipo **char**.

Nesta aula vamos aprender a definir estruturas de dados heterogêneas, isto é, podemos definir um tipo **novo** de estrutura em que misturamos diversos tipos. É um tipo novo no sentido que não existe na linguagem C original.

24.1 Um Tipo de Estrutura Simples: PESSOA

Para definir um tipo novo, usamos a declaração **typedef**. No primeiro exemplo a seguir, definimos um tipo novo chamado `PESSOA`, contendo o nome, a altura e o peso. Cada um destes três componentes é chamado de *campo* de um tipo. O campo Nome é um string (vetor de caracteres), e os campos Altura e Peso são do tipo **int**.

```
/* definição de tipo */
typedef struct{
char Nome[100]; /* declaração de um vetor de caracteres */
int Altura;  /* em centimetros */
int Peso;    /* em quilos */
} PESSOA;   /* declaração do tipo PESSOA */
```

A tabela a seguir ilustra os campos do tipo `PESSOA`.

Nome do campo	*tipo*
`Nome`	`string`
`Altura`	**int**
`Peso`	**int**

Para declarar uma variável do tipo `PESSOA`, deve-se ter a palavra `PESSOA` seguida pelos nomes das variáveis, como na linha a seguir, em que declaramos três variáveis do tipo `PESSOA`.

```
PESSOA marcio, alcides, maria; /* 3 vars do tipo PESSOA */
```

Para referenciar um determinado campo de variável, deve-se ter o nome da variável seguida de um ponto e o nome do campo. Por exemplo, `marcio.Altura` refere-se ao campo `Altura` da variável `marcio` (que é do tipo `PESSOA`). Nas linhas a seguir atribui-se o valor 170 ao campo `Altura` de `marcio`, e a seguir este valor é exibido através de `printf()`.

```
marcio.Altura=170;
printf("Altura: %d centimetros\n", marcio.Altura);
```

O campo `Nome` pode receber uma sequência de caracteres. Por exemplo, nas duas linhas a seguir o nome ”Marcio Oliveira”é atribuído ao campo `Nome` (usando a função `strcpy`[8]), que é exibido através de `printf()`. Note as aspas duplas em torno de Marcio Oliveira.

```
# include <string.h>
⋮
strcpy (marcio.Nome, "Marcio Oliveira");
printf("Nome: %s\n", marcio.Nome);
```

Alternativamente o campo `Nome`, ou `Altura`, pode ser lido através de `scanf()`.

[8]Detalhes sobre esta função, veja aula de strings.

```
printf ("Entre com um nome: ");
scanf ("%[^\n]", marcio.Nome);
printf("Nome: %s\n", marcio.Nome);
```

A variável `marcio` fica definida com os valores seguintes.

Nome do campo	*valor*
`Nome`	Marcio Oliveira
`Altura`	170
`Peso`	??? (lixo)

Com a linha:

```
PESSOA amigos[20]; /* grupo (vetor) de 20 pessoas */
```

declara-se um vetor chamado `amigos`, de 20 elementos do tipo `PESSOA`. Com as linhas:

```
1 amigos[0].Peso= 72;
2 printf("Peso do zero-esimo amigo: %d\n", amigos[0].Peso);
```

na Linha 1, o campo `Peso` da variável `amigos[0]` recebe o valor 72, que é exibido a seguir pelo `printf()`, na Linha 2.

Abaixo o programa completo deste exemplo simples.

```
1 /*
2  * programa de estrutura simples
3  * para ilustrar struct
4  *
5  */
6 # include <stdio.h>
7 # include <string.h>
8
9 int main(){
10   /* definição de tipo */
11   typedef struct{
12     char Nome[100]; /* declaração de um vetor de caracteres */
13     int Altura;  /* em centimetros */
14     int Peso;    /* em quilos */
15   } PESSOA;   /* declaração do tipo PESSOA */
16
17   /* note a ausencia da palavra struct nas linhas abaixo */
18
19   PESSOA marcio, alcides, maria; /* 3 vars do tipo PESSOA */
20   PESSOA amigos[20]; /* grupo (vetor) de 20 pessoas */
21
22   marcio.Altura=170;
23   printf("Altura: %d centimetros\n", marcio.Altura);
24   strcpy (marcio.Nome, "Marcio Oliveira");
25   printf("Nome: %s\n", marcio.Nome);
26   amigos[0].Peso= 72;
27   printf("Peso do zero-esimo amigo: %d\n", amigos[0].Peso);
28
29   /* saida do programa e':
30    * Altura: 170 centimetros
31    * Nome: Marcio Oliveira
32    * Peso do zero-esimo amigo: 72
33    */
34   return 0;
35 }
```

24.2 Um Tipo mais Elaborado: CARRO

Veremos a seguir um exemplo de tipo mais elaborado, para ilustrar que é possível definir um tipo **struct** dentro de outro tipo **struct**!

Utilizamos de novo a declaração **typedef** para declarar o tipo chamado `CARRO`. O **struct interno** chama-se `Endereco`, dentro do tipo `CARRO`, como se vê abaixo. O campo `Endereco` (que também é **struct**) possui os seguintes 5 campos: `RuaeNum`, `Bairro`, `CEP`, `Cidade`, `Telefone`, que devem constituir o endereço completo do dono de um carro. Todos os 5 campos são do tipo sequência de caracteres. Além de `Endereco`, os outros 10 campos do tipo `CARRO` são: `NomeDono`, `Modelo`, `Ano`, `Km`, `Fabricante`, `Cor`, `NumPortas`, `GasOuAlc`, `Preco`, `Chapa`.

```
1 # define MAX 100
2 typedef struct{ /* CARRO é o nome do tipo de estrutura */
3   char NomeDono [MAX];    /* string */
4   struct {   /* um struct dentro de outro */
5     char RuaeNum [MAX]; /* Nome da rua e numero, string */
6     char Bairro [MAX];
7     char CEP [MAX];
8     char Cidade [MAX];
9     char Telefone [MAX];
10  } Endereco;     /* nome de um campo, que é struct também */
11  char Modelo [MAX];
12  int Ano;
13  int Km;  /* quilometragem atual */
14  char Fabricante [MAX]; /* nome do fabricante */
15  char Cor [MAX];
16  int NumPortas; /* número de portas */
17  int GasOuAlc;  /* 1==gasolina, 2==alcool, 3==flex */
18  float Preco;  /* preço atual de mercado, em reais */
19  char Chapa [MAX];
20 } CARRO;   /* note o ; aqui. CARRO é o nome do tipo */
```

A tabela abaixo ilustra os vários campos do tipo `CARRO`.

Nome do campo	***tipo***		
`NomeDono`	`string`		
`Endereco`	**struct**	***Nome do campo***	***tipo***
		`RuaeNum`	`string`
		`Bairro`	`string`
		`CEP`	`string`
		`Cidade`	`string`
		`Telefone`	`string`
`Modelo`	`string`		
`Ano`	**int**		
`Km`	**int**		
`Fabricante`	`string`		
`Cor`	`string`		
`NumPortas`	**int**		
`GasOuAlc`	**int**		
`Preco`	**float**		
`Chapa`	`string`		

A linha:

```
1   CARRO meucarro, carronovo, carrodopai; /* 3 vars do tipo CARRO */
```

declara 3 variáveis do tipo `CARRO`.

As linhas:

```
# include <stdio.h>
# include <string.h>
⋮
  strcpy (meucarro.Endereco.RuaeNum, "Rua Padre Anchieta, 1011");
  strcpy (meucarro.Modelo, "Astro");
  meucarro.Ano = 2999;
  meucarro.Km = 31;
  strcpy (meucarro.Fabricante, "HN");
  strcpy (meucarro.Cor, "verde");
  meucarro.NumPortas = 2;
  meucarro.GasOuAlc = 1;
  meucarro.Preco= 432000.00;
  printf("Nome do meu modelo: %s\n", meucarro.Modelo); /* mostra Astro */
  printf("Nome da minha rua e numero: %s \n", meucarro.Endereco.RuaeNum);
```

definem e exibem alguns campos da variável `meucarro`.

A variável `meucarro` fica definida com os seguintes valores.

Nome do campo	***valor***		
`NomeDono`	??? (lixo)		
`Endereco`	**struct**	***Nome do campo***	***valor***
		`RuaeNum`	Rua Padre Anchieta, 1011
		`Bairro`	??? (lixo)
		`CEP`	??? (lixo)
		`Cidade`	??? (lixo)
		`Telefone`	??? (lixo)
`Modelo`	Astro		
`Ano`	2999		
`Km`	31		
`Fabricante`	HN		
`Cor`	verde		
`NumPortas`	2		
`GasOuAlc`	1		
`Preco`	432000.00		
`Chapa`	??? (lixo)		

A seguir o programa completo, com o tipo `CARRO`.

```c
# include <stdio.h>
# include <string.h>

# define MAX 100

/*
 * programa de carros em struct
 */

int main () {
  typedef struct { /* CARRO é o nome do tipo de estrutura */
    char NomeDono [MAX];    /* string */
    struct {   /* um struct dentro de outro */
      char RuaeNum [MAX]; /* Nome da rua e numero, string */
      char Bairro [MAX];
      char CEP [MAX];
      char Cidade [MAX];
      char Telefone [MAX];
    } Endereco;     /* nome de um campo, que é struct também */
    char Modelo [MAX];
    int Ano;
    int Km;  /* quilometragem atual */
    char Fabricante [MAX]; /* nome do fabricante */
    char Cor [MAX];
    int NumPortas; /* número de portas */
    int GasOuAlc;  /* 1==gasolina, 2==alcool, 3==flex */
    float Preco;  /* preço atual de mercado, em reais */
    char Chapa [MAX];
  } CARRO;   /* note o ; aqui. CARRO é o nome do tipo */

  CARRO meucarro, carronovo, carrodopai; /* 3 vars do tipo CARRO */

  strcpy (meucarro.Endereco.RuaeNum, "Rua Padre Anchieta, 1011");
  strcpy (meucarro.Modelo, "Astro");
  meucarro.Ano = 2999;
  meucarro.Km = 31;
  strcpy (meucarro.Fabricante, "HN");
  strcpy (meucarro.Cor, "verde");
  meucarro.NumPortas = 2;
  meucarro.GasOuAlc = 1;
  meucarro.Preco= 432000.00;
  printf("Nome do meu modelo: %s\n", meucarro.Modelo); /* mostra Astro */
  printf("Nome da minha rua e numero: %s \n", meucarro.Endereco.RuaeNum);

  /* saida é:
   * Nome do meu modelo: Astro
   * Nome da minha rua e numero: Rua Padre Anchieta, 1011
   */

  return 0;
}
```

24.3 Exercícios

1. Declarar um vetor chamado `frota` com até `NUM_MAX` elementos que sejam do tipo `CARRO`.
2. Atribuir (ou ler) os valores dos campos de todos os carros no vetor `frota`, de uma empresa locadora de carros. Se a frota for muito grande e não couber na memória interna (RAM), grave um único arquivo com todos os carros em algum meio magnético (pendrive ou HD). É um banco de dados.
3. Elabore um programa que ordena o vetor`frota`, ou o arquivo de carros, pelo número da placa.

4. Elabore um programa que lê um valor de placa, busca no vetor `frota`, ou no arquivo de carros, (por busca binária) um carro com esta placa, e lista todos os seus campos, se houver.
5. Elabore um banco de dados de uma video-locadora, com ordenação e busca binária.
6. É possível ter dois `structs` internos dentro de um **struct**?
7. É possível definir uma matriz, digamos 10 por 20, como campo de um **struct**?
8. Escrever um programa que calcula e imprime toda a tabuada de multiplicação de inteiros, utilizando **struct**.