

Técnicas de Programação II

Revisão TP1

Profa.: Leila Andrade

e-mail: leila@uniriotec.br

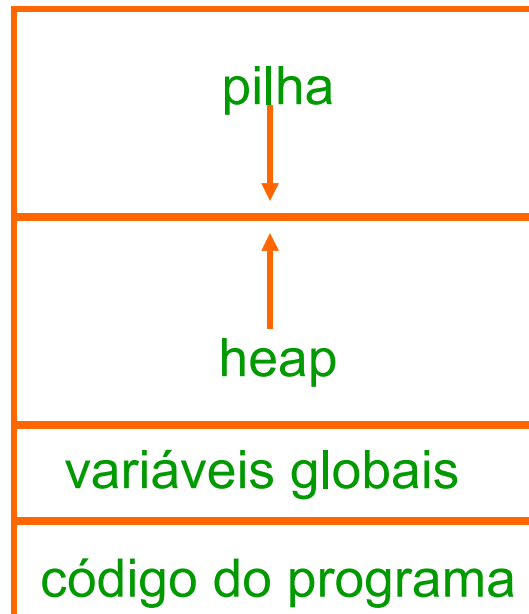
Material inspirado no curso TPII
da Profa. Adriana Alvim

O mapa de memória de C

- Um programa em **C compilado** cria e usa quatro regiões, logicamente distintas na memória, que possuem funções específicas
 - a memória que contém o código do seu programa
 - a região onde as **variáveis globais** são armazenadas
 - a região chamada de **pilha**
 - guarda o endereço de retorno das chamadas de função
 - argumentos para funções e variáveis locais
 - a região chamada de **heap**
 - região de memória livre
 - o programa pode usar via alocação dinâmica

O mapa de memória de C

- Embora a disposição física exata de cada uma das quatro regiões possa diferir entre tipos de CPU e implementações de C
 - o diagrama a seguir mostra conceitualmente como seu programa aparece na memória



Variáveis

- Uma variável representa um espaço na memória do computador para armazenar um determinado tipo de dado (valor)
 - que pode ser modificado pelo programa
- Em **C**, todas as variáveis devem ser declaradas

Variáveis

- Na declaração de uma variável, obrigatoriamente, devem ser especificado seu **tipo** e seu **nome**
 - o **nome** da variável serve de referência ao dado armazenado no espaço de memória da variável
 - e o **tipo** da variável determina a natureza do dado que será armazenado

Tipos básicos de dados

- Existem 5 tipos básicos de dados em C
 - caractere (`char`)
 - inteiro (`int`)
 - real (ponto flutuante) (`float`)
 - real (ponto flutuante) de precisão dupla (`double`)
 - sem valor (`void`)
- Todos os outros tipos de dados são baseados em um desses tipos

Tipos básicos de dados

- Exceto o tipo `void`, os tipos de dados básicos podem ter vários modificadores precedendo-os
 - um modificador é usado para alterar o significado de um tipo básico
- Lista dos modificadores
 - `signed` (com sinal)
 - `unsigned` (sem sinal)
 - `long`
 - `short`

Tipos básicos - valores inteiros

- Para armazenar **valores inteiros** existem os tipos básicos
 - char, int
- Aplicando os modificadores **long** e **short** temos ainda os tipos
 - short int, long int

Tipos básicos - valores inteiros

- Estes tipos diferem entre si pelo espaço de memória que ocupam e conseqüentemente pelo intervalo de valores que podem representar
 - o tipo `char`, por exemplo, ocupa 1 byte de memória (8 bits)
 - podendo representar $2^8 = 256$ valores distintos

Tipos básicos - valores inteiros

- Os tipos `short int` e `long int` podem ser referenciados simplesmente com `short` e `long`, respectivamente
- Na maioria das implementações da linguagem C, o tipo `short` é representado por `2 bytes` e o `long` por `4 bytes`
- O tipo `int` puro é, em geral, mapeado para o tipo inteiro natural da máquina, que pode ser `short` ou `long`
 - sua representatividade é maior ou igual a do tipo `short` e menor ou igual a do tipo `long`

Tipos básicos - valores inteiros

- Todos estes tipos podem ainda ser modificados para representar apenas valores positivos, o que é feito precedendo o tipo com o modificador “sem sinal”
`unsigned`
- O uso de `signed` com inteiros é permitido mas redundante
 - pois a declaração padrão de inteiros assume um número com sinal

Tipos básicos - valores inteiros

- A tabela abaixo compara os tipos para valores inteiros e suas representatividades

Tipo	Tamanho	Representatividade
char	1 byte	-128 a 127
unsigned char	1 byte	0 a 255
short int	2 bytes	-32.768 a 32.767
unsigned short int	2 bytes	0 a 65.535
long int	4 bytes	-2.147.483.648 a 2.147.483.647
unsigned long int	4 bytes	4.294.967.295

Tipos básicos - valores inteiros

- O tipo `char` costuma ser usado apenas para representar códigos de caracteres
 - como veremos mais adiante
- Na prática usamos o tipo `int`
 - sem modificadores
- para representar números inteiros

Tipos básicos - valores reais

- A linguagem oferece dois tipos básicos para a representação de números reais (ponto flutuante)
 - float e double
- Há duas notações para números reais
 - a notação decimal
 - e a notação exponencial

decimal	exponencial
317.42	$0.31742 * 10^3$
57325000000000	$0.57325 * 10^{14}$
0.000000161	$0.161 * 10^{-6}$

- em C $0.161 * 10^{-6}$ equivale a $0.161e-6$

Tipos básicos - valores reais

- A faixa dos tipos **float** e **double** é dada em dígitos significativos (**precisão**)
- As **grandezas** dos tipos **float** e **double** dependem do método usado para representar números em ponto flutuante
 - qualquer que seja o método
 - o número é muito grande!**
 - o padrão ANSI especifica que a faixa mínima é de **1E-37 a 1E+37**

Tipos básicos - valores reais

- A tabela abaixo mostra o número mínimo de dígitos significativos (**precisão**) e a grandeza (**representatividade**)

tipo	tamanho	precisão	representatividade
float	4 bytes	seis dígitos	(+/-) 10^{-38} a 10^{38}
double	8 bytes	dez dígitos	(+/-) 10^{-308} a 10^{308}
long double	12 bytes	dez dígitos	

- se você precisa de um número maior do que **unsigned long int** deverá usar um tipo **float** ou **double**, e perderá precisão

Declaração de variáveis

- Para armazenar um dado (**valor**) na memória do computador
 - deve-se reservar o espaço correspondente ao tipo do dado (**char, int, float, etc..**) a ser armazenado
- A declaração de uma variável reserva um espaço na memória para armazenar um dado do **tipo** da variável e associa o **nome da variável** a este espaço de memória

Declaração de variáveis

```
int a; /* declara uma variável do tipo int */  
int b; /*declara outra variável do tipo int*/  
float c; /*declara uma variável do tipo float*/
```

- Atribui-se valor a uma variável utilizando-se o operador de atribuição =

```
a = 5; /* armazena o valor 5 em a */  
b = 10; /* armazena o valor 10 em b */  
c = 5.3; /* armazena o valor 5.3 em c */
```

Declaração de variáveis

- A linguagem permite que variáveis de mesmo tipo sejam declaradas juntas

```
int a, b; /* declara duas variáveis do tipo int*/
```

- Estes valores devem ter o mesmo tipo da variável
- Não é possível, por exemplo, armazenar um número real numa variável do tipo `int`

```
int a;
```

```
a = 4.3; /* a variável armazenará o valor 4 */
```

- será armazenada em `a` apenas a parte inteira do número real, isto é, `4`

Declaração de variáveis

- Em **C**, as variáveis podem ser inicializadas na declaração
- Por exemplo

```
int a = 5, b = 10; /* declara e inicializa as  
    variáveis */  
float c = 5.3;
```

Valores constantes

- Nos nossos códigos também usamos valores constantes
- Seja a atribuição
 $a = b + 123;$
 - sendo a e b variáveis supostamente já declaradas, reserva-se um espaço para armazenar a constante 123
 - para que a operação possa ser avaliada em tempo de execução
 - esse valor constante está armazenado em um espaço de memória próprio
 - no caso, a constante é do tipo inteiro, então um espaço de quatro bytes (em geral) é reservado e o valor 123 é armazenado nele

Valores constantes

- A diferença básica em relação às variáveis, como os nomes dizem (**variáveis** e **constantes**), é que o valor armazenado numa área de constante não pode ser alterado
- Constantes em **C** podem ser de qualquer um dos tipos de dados básicos (e seus modificadores)
- A maneira como cada constante é representada depende do seu tipo
- Constantes do tipo **caractere** são envolvidas por aspas simples

'a', 'B', ':', '+', '\n', '\0', '\", '\\', \' '

Valores constantes

- Constantes **inteiras** são especificadas como números sem componentes fracionários
- Por padrão, o compilador **C** encaixa uma constante numérica no menor tipo de dado compatível que pode contê-lo
- Assim **10** é um **int** por padrão
 - mas **60.000** é um **unsigned**
 - e **100.000** é **long**

Valores constantes

- Constantes do tipo real
 - uma constante real deve ser escrita com um ponto decimal ou valor de expoente
 - sem nenhum sufixo, uma constante real é do tipo `double`
 - exceção para a regra do menor tipo!
 - se quisermos uma constante `real` do tipo `float`, devemos, a rigor, acrescentar o sufixo `F` ou `f`

Valores constantes

- É possível especificar precisamente o tipo da constante numérica utilizando um **sufixo**

Tipo de dado	Exemplos de constantes
int	1 123 21000 -234
long int	35000L -34L
short int	10 -12 90
unsigned int	10000U 987U 40000
float	123.23F 4.34e-3F
double	123.23 12312333 -0.9876324
long double	1001.2L

Variáveis com valores indefinidos

- Um dos erros comuns em programas de computador é o uso de variáveis cujos valores ainda estão indefinidos
- Por exemplo, o trecho de código abaixo está errado, pois o valor armazenado na variável **b** está indefinido e tentou-se usá-lo na atribuição a **c**
- É comum dizer que **b** tem “lixo”

```
int a, b, c;
```

```
a = 2;
```

```
c = a + b; /* ERRO: b tem "lixo" */
```

Variáveis com valores indefinidos

- Alguns destes erros são óbvios (como o ilustrado acima) e o compilador é capaz de nos reportar uma advertência
 - no entanto, muitas vezes o uso de uma variável não definida fica difícil de ser identificado no código
- É um erro comum em programas e uma razão para alguns programas funcionarem na parte da manhã e não funcionarem na parte da tarde

Variáveis com valores indefinidos

- Todos os erros em computação têm lógica
- A razão de o programa poder funcionar uma vez e não funcionar outra é que, apesar de indefinido, o valor da variável existe
- No caso acima, pode acontecer que o valor armazenado na memória ocupada por `b` seja `0`, fazendo com que o programa funcione
- Por outro lado, pode acontecer de o valor ser `-293423` e o programa não funcionar

Operadores

- C é muito rica em operadores
- C define quatro classes de operadores
 - aritméticos
 - relacionais
 - lógicos
 - bit a bit

Operadores aritméticos

- Os operadores aritméticos **binários** são
 $+$, $-$, $*$, $/$ e o operador módulo $\%$
há ainda o operador unário $-$
- A operação é feita na precisão dos operandos
 - a expressão $5/2$ resulta no valor 2 , pois a operação de divisão é feita em precisão **inteira**, já que os dois operandos (5 e 2) são constantes inteiras
 - a divisão de inteiros trunca a parte fracionária, pois o valor resultante é sempre do mesmo tipo da expressão
 - a expressão $5.0/2.0$ resulta no valor real 2.5 pois a operação é feita na precisão **real** (double, no caso)

Operadores aritméticos

- O operador módulo, %, não se aplica a valores reais, seus operandos devem ser do **tipo inteiro**
 - este operador produz o resto da divisão do primeiro pelo segundo operando
- Como exemplo de aplicação deste operador, podemos citar o caso em que desejamos saber se o valor armazenado numa determinada variável inteira **x** é par ou ímpar
- Para tanto, basta analisar o resultado da aplicação do operador %, aplicado à variável e ao valor **dois**
 - x % 2** se resultado for **zero**, o número é **par**
 - x % 2** se resultado for **um**, o número é **ímpar**

Operadores aritméticos

- O operador - **unário** tem precedência maior que *****, **/** e **%**
 - este operador multiplica seu único operando por -1
 - isto, o número troca de sinal
- Os operadores *****, **/** e **%** têm precedência maior que os operadores **+** e **-**

Operadores aritméticos

- Operadores com mesma precedência são avaliados da esquerda para a direita
- Assim, na expressão
$$a + b * c / d$$
 - executa-se primeiro a multiplicação
 - seguida da divisão
 - seguida da soma
- É possível utilizar parênteses para alterar a ordem de avaliação de uma expressão
 - para avaliar a soma primeiro, escreve-se
$$(a + b) * c / d$$

Operadores de atribuição

- Na linguagem **C**, uma atribuição é uma expressão cujo valor resultante corresponde ao valor atribuído
- A atribuição
 - a = 5;**
 - resulta no valor **5** além, é claro, de armazenar o valor **5** na variável **a**
- A ordem de avaliação é da direita para a esquerda
 - y = x = 5;**
 - o computador avalia **x = 5**, armazenando **5** em **x**, e em seguida armazena em **y** o valor produzido por **x = 5**, que é **5**
- Portanto, ambos, **x** e **y**, recebem o valor **5**

Operadores de atribuição

- A linguagem também permite utilizar os chamados operadores de atribuição compostos
 - `i = i + 2;`
 - em que a variável à esquerda do sinal de atribuição também aparece à direita
 - podem ser escritas de forma mais compacta
 - `i += 2;`
 - usando o operador de atribuição composto `+=`
- Analogamente, existem, entre outros, os operadores de atribuição
 - `-=`, `*=`, `/=`, `%=`

Operadores de atribuição

- De forma geral, comandos do tipo
`variável op= expressão;`
 - são equivalentes a
`variável = variável op (expressão);`
- Note a presença dos parênteses em torno de expressão
 - assim
`x *= y + 1;`
 - equivale a
`x = x * (y + 1)`
 - e não a
`x = x * y + 1;`

Operadores de incremento e decremento

- A linguagem **C** apresenta ainda dois operadores não convencionais
- São os operadores de **incremento** e **decremento**, que possuem precedência comparada ao **-** unário e servem para incrementar e decrementar uma unidade nos valores armazenados nas variáveis
- Se **n** é uma variável que armazena um valor, o comando

`n++;`

- incrementa de uma unidade o valor de **n**
 - análogo para o decremento em `n--`

Operadores de incremento e decremento

- O aspecto não usual é que `++` e `--` podem ser usados tanto como
 - operadores pré-fixados
 - antes da variável, como em `++n`
 - pós-fixados
 - após a variável, como em `n++`
- Em ambos os casos, a variável `n` é incrementada
 - a expressão `++n` incrementa `n` antes de usar seu valor
 - enquanto `n++` incrementa `n` após seu valor ser usado

Operadores de incremento e decremento

- Isto significa que, num contexto onde o valor de `n` é usado, `++n` e `n++` são diferentes
 - Se `n` armazena o valor `5`, então
 - `x = n++;` atribui `5` a `x`, mas:
 - `x = ++n;`
atribuiria `6` a `x`
 - em ambos os casos, `n` passa a valer `6`, pois seu valor foi incrementado em uma unidade
 - os operadores de incremento e decremento podem ser aplicados somente em variáveis
 - uma expressão do tipo `x = (i + 1)++` é ilegal
- `777++ /* constante não pode ser incrementada */`

Operadores de incremento e decremento

- A linguagem **C** oferece diversas formas compactas para se escrever um determinado comando
- Neste curso, a idéia é evitar as formas compactas pois elas tendem a dificultar a compreensão do código
- Mesmo para programadores experientes, o uso das formas compactas deve ser feito com critério

Operadores de incremento e decremento

- Por exemplo, os comandos

```
a = a + 1;
```

```
a += 1;
```

```
a++;
```

```
++a;
```

- são todos equivalentes e o programador deve escolher o que achar mais adequado e simples
- em termos de desempenho, qualquer compilador razoável é capaz de otimizar todos estes comandos da mesma forma

Operadores relacionais e lógicos

- Os operadores relacionais em C são
 - < menor que
 - > maior que
 - <= menor ou igual que
 - >= maior ou igual que
 - == igual a
 - != diferente de
- Estes operadores comparam dois valores
- O resultado produzido por um operador relacional é zero (**falso**) ou um (**verdadeiro**)

Operadores lógicos

- Os operadores lógicos combinam expressões booleanas
- A linguagem oferece os seguintes operadores lógicos
 - && operador binário E (AND)
 - || operador binário OU (OR)
 - ! operador unário de NEGAÇÃO (NOT)
- Expressões conectadas por && ou || são avaliadas da esquerda para a direita, e a avaliação pára assim que a veracidade ou falsidade do resultado for conhecida
- Recomenda-se o uso de parênteses em expressões que combinam operadores lógicos e relacionais

Operadores lógicos

- Os operadores relacionais e lógicos são normalmente utilizados para tomada de decisões
- No entanto, pode-se utilizá-los para atribuir valores a variáveis

Operadores lógicos

- O trecho de código abaixo é válido e armazena o valor 1 em `a` e 0 em `b`

```
int a, b;
```

```
int c = 23;
```

```
int d = c + 4;
```

```
a = (c < 20) || (d > c); /* verdadeiro */
```

```
b = (c < 20) && (d > c); /* falso */
```

- Na avaliação da expressão atribuída à variável `b`, a operação `(d>c)` não chega a ser avaliada, pois independente do seu resultado a expressão como um todo terá como resultado 0 (falso), uma vez que a operação `(c<20)` tem valor falso

Operadores lógicos

- Embora **C** não tenha um operador lógico **OR** exclusivo (**XOR**)
 - você pode criar uma função que executa esta tarefa usando os outros operadores lógicos
 - o resultado de uma operação **XOR** é verdadeiros se, e somente se, um operando (mas não os dois) for verdadeiro

Operadores lógicos

```
#include <stdio.h>
int xor(int a, int b);
int main() {
    printf("\n%d", xor(1,0));
    printf("\n%d", xor(1,1));
    printf("\n%d", xor(0,1));
    printf("\n%d", xor(0,0));
    return 0;
}
```

```
/* executa uma operação lógica XOR usando os dois
argumentos */
```

```
int xor(int a, int b)
{
    return (a || b) && !(a && b);
}
```

Avaliação de expressões

- Note também que as duas expressões

`a == b` e `a = b`

- são parecidas na forma porém possuem funções completamente distintas
- a expressão `a == b` é um teste de igualdade, enquanto que `a = b` é uma atribuição
- escrever

`if (a = 1) ... ERRO!`

- ao invés de

`if (a == 1) ...`

- é um erro de programação comum, a expressão do primeiro `if` é sempre verdadeira

Operador `sizeof`

- Outro operador fornecido por `C`, `sizeof`, resulta no número de `bytes` de um determinado tipo
- Por exemplo

```
int a = sizeof(float) ;
```

 - armazena o valor `4` na variável `a`, pois um `float` ocupa `4 bytes` de memória
- Este operador pode também ser aplicado a uma variável ou expressão, como por exemplo `a+b`, retornando o número de bytes do tipo associado à variável ou à expressão

Operador sizeof

```
/* Calcula o tamanho de alguns tipos basicos */
# include <stdio.h>
int main()
{
    printf("Tamanho de alguns tipos basicos.\n\n");
    printf("      char:%3d bytes \n", sizeof(char));
    printf("      short:%3d bytes \n", sizeof(short));
    printf("      int:%3d bytes \n", sizeof(int));
    printf("      Long:%3d bytes \n", sizeof(long));
    printf("      unsigned:%3d bytes \n", sizeof(unsigned));
    printf("      float:%3d bytes \n", sizeof(float));
    printf("      double:%3d bytes \n", sizeof(double));
    printf(" Long double:%3d bytes \n", sizeof(long double));
    return 0;
}
```

Conversão de tipos em expressões

- Uma **expressão aritmética**, como por exemplo $a + b$, possui um **valor** e um **tipo**
- Se ambos, a e b , são do mesmo tipo, inteiro por exemplo, a expressão também é do tipo inteiro
- Entretanto, se a e b são de tipos diferentes, então $a + b$ é uma expressão mista
- Neste caso a linguagem faz **C** faz uma conversão automática de valores na avaliação da expressão
 - note que o valor das variáveis, da forma como foram armazenados na memória, não são alterados
 - é apenas um cópia temporária de x que é convertida durante o cálculo da expressão

Conversão de tipos em expressões

- Em geral, o tipo menor é promovido para o tipo maior antes que a operação prossiga
 - o resultado é do tipo maior

Conversão de tipos em expressões

- Se não houver operandos **unsigned**, o seguinte conjunto informal de regras será suficiente
 - se um dos operandos for **long double**
 - converte o outro para **long double**
 - senão, se um dos operandos for **double**
 - converte o outro para **double**
 - senão, se um dos operandos for **float**
 - converte o outro para **float**
 - senão, converte **char** e **short** para **int**
 - então, se um dos operandos for **long**
 - converte o outro para **long**

Conversão de tipos em expressões

- Uma vez que essas regras tenham sido aplicadas
 - cada par de operandos é do mesmo tipo
 - e o resultado de cada operação é do mesmo tipo de ambos os operandos
- Assim, na expressão $3/1.5$
 - o valor da constante 3 (tipo `int`) é promovido (convertido) para `double`
 - antes de a expressão ser avaliada
 - pois o segundo operando é do tipo `double` (1.5)
 - e a operação é feita na precisão do tipo mais representativo

Conversão de tipos em atribuições

- Além da conversão automática **em expressões** de tipo misto
- Quando, numa **atribuição**, o tipo do valor atribuído é diferente do tipo da variável, também há uma conversão automática de tipo
- Em um comando de atribuição, a regra de conversão de tipos é simples
 - o valor do lado direito (o lado da expressão) de uma atribuição é convertido no tipo do lado esquerdo (a variável destino)

Conversão de tipos em atribuições

- Por exemplo

```
int a = 3.5;
```

- o valor 3.5 é convertido para inteiro (isto é, passa a valer 3) antes de a atribuição ser efetuada
- como resultado, o valor atribuído à variável é 3 (inteiro)

```
double d; int i; d = i;
```

- faz com que o valor de `i`, que é inteiro, seja convertido para `double` e depois atribuído à variável `d`

Conversão de tipo

- Alguns compiladores exibem advertências quando a conversão de tipo pode significar uma perda de precisão
 - é o caso da conversão **real** para **inteiro**, por exemplo
- O programador pode explicitamente requisitar uma conversão de tipo através do uso do operador de molde de tipo (operador **cast**)
- A forma genérica é
(tipo) expressão
- onde tipo é qualquer tipo de dados válido em **C**

Conversão de tipo

- Por exemplo, são válidos (e isentos de qualquer advertência por parte dos compiladores) os comandos abaixo

```
int a, b;
```

```
a = (int) 3.5;
```

```
b = (int) 3.5 % 2;
```

- que é equivalente a `((int) 3.5) % 2` pois o operador `cast` tem precedência maior do que o operador `%`

Tabela da hierarquia dos operadores

- Precedência em ordem decrescente

Operadores	Associatividade
()	esquerda para direita
! ++ -- - +(unário) sizeof(tipo)	direita para esquerda
* / % (unário)	esquerda para direita
+ -	esquerda para direita
< <= > >=	esquerda para direita
== !=	esquerda para direita
&&	esquerda para direita
	esquerda para direita
= += -= *= /=	direita para esquerda

Entradas e saídas básicas

- A linguagem **C** não possui comandos de entrada e saída de dados
- Tudo em **C** é feito através de funções, inclusive as operações de entrada e saída
- Por isso, já existe em **C** uma biblioteca padrão que possui as funções básicas normalmente necessárias
- Nela podemos, por exemplo, encontrar funções matemáticas do tipo raiz quadrada, seno, co-seno, etc., funções para a manipulação de cadeias de caracteres e **funções de entrada e saída**

Entradas e saídas básicas

- A seguir, serão apresentadas as duas funções básicas de **entrada** e **saída** disponibilizadas pela biblioteca padrão
- Para utilizá-las, é necessário incluir o **protótipo** destas funções no código
 - este assunto será tratado em detalhes na seção sobre funções
- Por ora, basta saber que é preciso escrever

```
#include <stdio.h>
```
- no início do programa que utiliza as funções da biblioteca de **entrada** e **saída**

Função `printf`

- A função `printf` possibilita a saída de valores (sejam eles `constantes`, `variáveis` ou resultado de `expressões`) segundo um determinado formato
- Informalmente, pode-se dizer que a forma da função é
`printf (cadeia de controle, outros argumentos) ;`
 - onde outros argumentos pode ser
 - lista de constantes, variáveis, expressões...

Função `printf`

- O primeiro parâmetro é uma cadeia de caracteres
 - em geral delimitada com aspas
 - que especifica o formato de saída das constantes, variáveis e expressões listadas em seguida
- Para cada valor que se deseja imprimir, deve existir um **especificador de formato** correspondente na **cadeia de controle**
- Os **especificadores de formato** variam com o tipo do valor e a precisão em que queremos que eles sejam impressos

Função `printf`

- Um **especificador de formato** é uma cadeia de caracteres precedida pelo símbolo percentual (%) e finalizada com um **caractere de conversão**

Função printf

- A seguir uma lista de alguns caracteres de conversão

caractere conversão	como o argumento convertido é impresso
<code>%c</code>	especifica um char
<code>%d</code>	especifica um int
<code>%u</code>	especifica um unsigned int
<code>%f</code>	especifica um double (ou float). Exemplo <code>7.123000</code>
<code>%e</code>	especifica um double (ou float) no formato científico (notação exponencial). Exemplo: <code>7.123000e+00</code>
<code>%g</code>	no formato <code>%f</code> ou <code>%e</code> , o que for menor
<code>%s</code>	especifica uma cadeia de caracteres

Função `printf`

- Alguns exemplos

```
printf ("%d %g\n", 33, 5.3);
```

- tem como resultado a impressão da linha

```
33 5.3
```

- ou

```
printf ("inteiro = %d Real = %g\n", 33, 5.3);
```

- com saída

```
inteiro = 33 Real = 5.3
```

- além dos especificadores de formato, podemos incluir textos no formato
- que são mapeados diretamente para a saída

Função `printf`

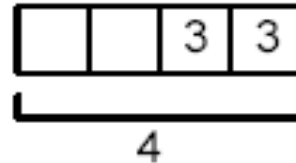
- Existem alguns caracteres especiais que são freqüentemente utilizados nos formatos de saída

<code>\n</code>	caractere de nova linha
<code>\t</code>	caractere de tabulação
<code>\r</code>	caractere de retrocesso (volta ao início da linha)
<code>\"</code>	o caractere <code>"</code>
<code>\\</code>	o caractere <code>\</code>
<code>%%</code>	o caractere <code>%</code>

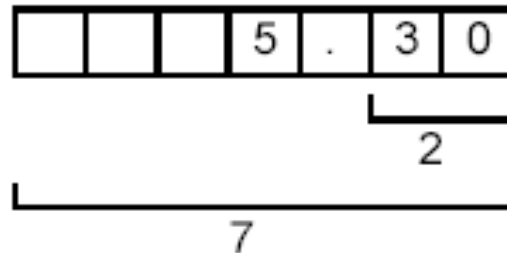
Função printf

- É possível também especificar o tamanho dos campos

`%4d`



`%7.2f`



- se quisermos fixar em 2 o número de casas decimais usadas para exibir valores reais, podemos usar o especificador de formato `%.2f`

Função `scanf`

- A função `scanf` permite capturar valores fornecidos via teclado pelo usuário do programa
 - e armazená-los em variáveis do nosso programa
- Informalmente, pode-se dizer que sua forma geral é
`scanf (cadeia de controle, outros argumentos...);`
- A `cadeia de controle` deve possuir especificadores de tipos similares aos mostrados para a função `printf`

Função scanf

- Para a função **scanf**, no entanto, existem especificadores diferentes para o tipo **float** e o tipo **double**

%c	especifica um char
%d	especifica um int
%u	especifica um unsigned int
%f, %g, %e	especificam um float
%lf, %lg, %le	especificam um double
%s	especifica uma cadeia de caracteres

Função `scanf`

- A principal diferença é que o formato deve ser seguido por uma lista de endereços de variáveis
 - na função `printf` passamos os valores de constantes, variáveis e expressões
- Na seção sobre ponteiros, este assunto será tratado em detalhes
 - por ora, basta saber que, para ler um valor e atribuí-lo a uma variável, deve-se passar o endereço da variável para a função `scanf`

Função `scanf`

- O operador `&` retorna o endereço de uma variável
 - assim, para ler um inteiro, deve-se ter

```
int n;  
scanf ("%d", &n);
```
 - dessa forma, o valor inteiro digitado pelo usuário é armazenado na variável `n`
 - para a função `scanf`, os especificadores `%f`, `%e` e `%g` são equivalentes

Função `scanf`

- Aqui, caracteres diferentes dos especificadores no formato servem para cercar a entrada

```
scanf ("%d:%d", &h, &m) ;
```

 - obriga que os valores (inteiros) fornecidos sejam separados pelo caractere dois pontos (:)
- Um espaço em branco dentro do formato faz com que sejam "pulados" eventuais brancos da entrada
 - os especificadores `%d`, `%f`, `%e` e `%g` automaticamente pulam os brancos que precederem os valores numéricos a serem capturados

Exemplo

- Para exemplificar o uso das funções de entrada e saída e a construção de expressões
 - vamos considerar um exemplo em que desejamos converter a altura de uma pessoa
 - dada em metros
 - para a altura expressa em pés e polegadas
 - sabe-se que um pé tem 30,48 cm e que 1 polegada tem 2,54 cm
 - assim, se o usuário entrar com o valor 1.8 (em metros)
 - o programa deve exibir o valor
5ft 10.9pol

Exemplo

```
/* Programa para converter altura em metros para ft e pol */
#include <stdio.h>
int main(void) {
    int f;      /* numero de pes */
    float p;    /* numero de polegadas */
    float h;    /* altura em metros */

    /* captura altura em metros */
    printf("Digite altura em metros: ");
    scanf("%f", &h);

    /* Calcula altura em pes e polegadas */
    h = 100*h; /* converte para centimetros */
    f = (int) (h/30.48); /* calcula numero de pes */
    p = (h-f*30.48)/2.54; /* calcula numero de polegas do restante */

    /* Exibe altura convertida */
    printf("Altura: %dft %.1fpol\n", f, p);

    return 0;
}
```

Controle de fluxo

- Até agora trabalhamos com programas formados por seqüências simples de comandos
- Para a construção de programas mais elaborados, precisamos ter acesso a mecanismos que permitam controlar o fluxo de execução dos comandos
 - é fundamental ter meios para tomar decisões que se baseiem em condições avaliadas em tempo de execução
 - também precisamos de mecanismos para a construção de procedimentos iterativos
 - procedimentos que repetem a execução de uma seqüência de comandos um determinado número de vezes

Controle de fluxo

- A linguagem **C** provê as construções fundamentais de controle de fluxo necessárias para programas bem estruturados
 - agrupamentos de comandos
 - tomadas de decisão (**if-else**)
 - laços com teste de encerramento no início (**while**, **for**) ou no fim (**do-while**)
 - e seleção de um dentre um conjunto de possíveis casos (**switch**)

Verdadeiro e Falso em C

- Muitos comandos em **C** contam com um teste condicional que determina o curso da ação
 - uma expressão condicional chega a um valor **verdadeiro** ou **falso**
 - um valor verdadeiro é **qualquer valor diferente de zero** (incluindo números negativos)
 - um valor falso é **zero**

Estruturas de bloco

- Cada chave aberta e fechada em **C** representa um bloco
- **Blocos de comando** são um grupo de comandos relacionados que são tratados como uma unidade
 - uma série de declarações ou comandos entre chaves
- Em **C**, onde é sintaticamente correto existir um comando, é correto existir um bloco de comandos

Estruturas de bloco

- Geralmente, o **bloco de comandos** é executado, processando-se cada declaração e comando em ordem, um de cada vez
 - a execução termina quando a última declaração ou comando foi executado
 - é possível sair de um bloco de comandos antes do seu término usando os comandos **goto**, **break**, **return** ou **continue**
 - também é possível entrar em um bloco de comandos sem ser no seu início usando os comandos **goto**, ou **switch**

Estruturas de bloco

- As declarações de variáveis só podem ocorrer no início do corpo da função ou no início de um **bloco**, isto é, devem seguir uma chave aberta
- Uma variável declarada dentro de um **bloco** é válida apenas dentro do **bloco**
- Após o término do **bloco**, a variável deixa de existir

Estruturas de bloco

- Por exemplo

```
...  
if ( n > 0 ){  
    int i;  
    ...  
}
```

```
... /* a variável i não existe neste ponto do  
    programa */
```

- A variável **i**, definida dentro do bloco do **if**, só existe dentro deste bloco
- É uma boa prática de programação declarar as variáveis o mais próximo possível dos seus usos

Tomada de decisão na linguagem C

- Sua forma pode ser

```
if (expressão) {  
    bloco de comandos 1  
    ...  
}
```

- Se **expressão** produzir um valor diferente de **0** (**verdadeiro**), o **bloco de comandos 1** será executado, caso contrário o controle passa para o próximo comando

Tomada de decisão na linguagem C

```
if (nota >= 90)
    printf("Parabens!");
printf("Sua nota %d", nota);
```

- Uma mensagem de parabéns é impressa somente quando a nota for maior ou igual a 90
- O segundo `printf()` é sempre executado
- Geralmente a **expressão** em um comando `if` envolve **operadores relacionais** ou **lógicos**, porém a sintaxe permite qualquer tipo de expressão

Tomada de decisão na linguagem C

- Exemplos

```
if (y != 0.0)
    x = x/y;
```

```
if (c == ' ') {
    brancos = brancos + 1;
    printf("Encontrou outro branco\n");
}
```

mas não!

```
if c == a
    area = area * a; /* falta parênteses */
```

Tomada de decisão na linguagem C

- Outra forma do `if` pode ser

```
if ( expressão ) {  
    bloco de comandos 1  
    ...  
}  
else {  
    bloco de comandos 2  
    ...  
}
```

Tomada de decisão na linguagem C

- Se **expressão** produzir um valor diferente de **0** (**verdadeiro**), o **bloco de comandos 1** será executado
- A inclusão do **else** requisita a execução do **bloco de comandos 2** se a expressão produzir o valor **0** (**falso**)
- Cada bloco de comandos deve ser delimitado por uma **chave aberta** e uma **chave fechada**

Tomada de decisão na linguagem C

- Se dentro de um **bloco** tivermos apenas um comando a ser executado, as chaves podem ser omitidas (na verdade, deixamos de ter um bloco)

```
if ( expressão )  
    comando1 ;  
else  
    comando2 ;
```

Tomada de decisão na linguagem C

- Em **C**, um **else** é associado ao último **if** que não tiver seu próprio **else**
- Para os casos em que a associação entre **if** e **else** não está clara
 - recomenda-se a criação explícita de blocos
 - mesmo contendo um único comando
- Se reescrevermos o programa, podemos obter o efeito desejado

Tomada de decisão

- Essa regra de associação do **else** propicia a construção do tipo **else-if**, sem que se tenha o comando **elseif** explicitamente na gramática da linguagem
- Em **C**, constrói-se estruturas **else-if** com **if**'s aninhados
- A seguir um exemplo

Tomada de decisão

```
/* temperatura (versao 3) */
#include <stdio.h>
int main (void){
    int temp;
    printf("Digite a temperatura: ");
    scanf("%d", &temp);
    if (temp < 10)
        printf("Temperatura muito fria \n");
    else if (temp < 20)
        printf(" Temperatura fria \n");
    else if (temp < 30)
        printf("Temperatura agradavel \n");
    else
        printf("Temperatura muito quente \n");
    return 0;
}
```

Estruturas de bloco

- Observamos que uma função `C` é composta por estruturas de blocos
- Cada chave aberta e fechada em `C` representa um bloco
- As declarações de variáveis só podem ocorrer no início do corpo da função ou no início de um `bloco`, isto é, devem seguir uma chave aberta
- Uma variável declarada dentro de um `bloco` é válida apenas dentro do `bloco`
- Após o término do `bloco`, a variável deixa de existir

Estruturas de bloco

- Por exemplo

```
...  
if ( n > 0 ){  
    int i;  
    ...  
}
```

```
... /* a variável i não existe neste ponto do  
    programa */
```

- A variável **i**, definida dentro do bloco do **if**, só existe dentro deste bloco
- É uma boa prática de programação declarar as variáveis o mais próximo possível dos seus usos

Operador condicional

- C possui também um operador ternário
 - requer três operandos
- chamado **operador condicional**
- Trata-se de um operador que substitui construções do tipo

```
...  
if ( a > b )  
    maximo = a;  
else  
    maximo = b;  
...
```

Operador condicional

- Sua forma geral é

`expressão1 ? expressão2 : expressão3;`

- a `expressão1` é avaliada
- se `expressão1` for verdadeira (diferente de zero), a `expressão2` é avaliada, e seu valor, convertido para o tipo da expressão, é o valor da expressão condicional
 - a `expressão3` não é avaliada
- caso contrário, avalia-se a `expressão3`, e seu valor, convertido para o tipo da expressão, é o valor da expressão condicional
 - a `expressão2` não é avaliada

Operador condicional

- O comando

```
maximo = a > b ? a : b ;
```

- substitui a construção com **if-else** mostrada anteriormente

```
...
```

```
if ( a > b )  
    maximo = a;  
else  
    maximo = b;
```

```
...
```

Tabela da hierarquia dos operadores

- Precedência em ordem decrescente

Operadores	Associatividade
()	esquerda para direita
! ++ -- - +(unário) sizeof(tipo)	direita para esquerda
* / % (unário)	esquerda para direita
+ -	esquerda para direita
< <= > >=	esquerda para direita
== !=	esquerda para direita
&&	esquerda para direita
	esquerda para direita
?:	direita para esquerda
= += -= *= /=	direita para esquerda

Operador vírgula

- O operador binário **vírgula** possui a mais baixa precedência entre todos os operadores da linguagem **C**
 - é usado para encadear diversas expressões

Operador vírgula

- Seja a expressão

expressão1, expressão2, expressão3, expressão4;

- as sub-expressões expressão1, expressão2, expressão3 e expressão4 são avaliadas da esquerda para a direita
- o **valor** da expressão e o **tipo** da expressão é o da última sub-expressão, ou seja, é o valor e o tipo de expressão4
- com outras palavras, o lado esquerdo de um operador vírgula é sempre avaliado como **void**
- significa que a expressão do lado direito torna-se o valor de toda a expressão separada por vírgulas

Operador vírgula

- O comando

```
r = (a, b, ..., c);
```

```
/* note que os parênteses são necessários*/
```

- é equivalente a

```
a; b; ... r = c;
```

- Exemplos

```
int a, b, c, x;
```

```
x = (a = 2, b = 3, c = 41);
```

```
printf ("a:%d b:%d c:%d x:%d", a, b, c, x);
```

```
– Resultado: a:2 b:3 c:41 x:41
```

Construções com laços

- Em programas computacionais, procedimentos iterativos são muito comuns
 - procedimentos que devem ser executados em vários passos
- Como exemplo, vamos considerar o cálculo do valor do fatorial de um número inteiro não negativo
- Por definição
$$n! = n \times (n-1) \times (n-2) \times \dots \times 3 \times 2 \times 1, \text{ onde } 0! = 1$$

Construções com laços

- Para calcular o fatorial de um número com um programa de computador
 - normalmente utilizamos um processo iterativo, em que o valor da variável varia de 1 a n
- A linguagem C oferece várias construções possíveis para a realização de laços iterativos
 - o primeiro a ser apresentado é o comando **while**
- Sua forma geral é

```
while (expressão) {  
    bloco de comandos  
    ...  
}
```

Comando `while`

```
while (expressão) {  
    bloco de comandos  
    ...  
}
```

- Se **`expressão`** resultar em verdadeiro
 - o bloco de comandos é executado
 - ao final do bloco, a expressão volta a ser avaliada e
- Enquanto **`expressão`** resultar em verdadeiro
 - o bloco de comandos (ou o corpo do laço) é executado repetidamente
- Quando **`expressão`** for avaliada em falso
 - o bloco de comandos deixa de ser executado
 - e a execução do programa prossegue com a execução dos comandos subsequentes ao bloco

Comando `while`

- A execução do comando `while` termina quando `expressão` é avaliada em falso
 - ou quando o controle é transferido para fora do bloco do `while` por um comando de `return`, `goto` ou `break`
 - o comando `continue` também pode alterar a execução do `while`

Comando `while`

- Uma possível implementação do cálculo do fatorial usando `while` é mostrada a seguir

```
/* Fatorial */
#include <stdio.h>
int main (void) {
    int i, n, f = 1;
    printf("Digite um número inteiro nao negativo:");
    scanf("%d", &n);
    i = 1;
    while (i <= n) {
        f = f * i;
        i = i + 1;
    }
    printf(" Fatorial = %d \n", f);
    return 0;
}
```

Comando `for`

- Uma segunda forma de construção de laços em **C**
 - mais compacta e amplamente utilizada, é com laços **for**
- A forma geral do **for** é

```
for (expr_inicial; expr_booleana; expr_de_incremento) {  
    bloco de comandos  
    ...  
}
```

Comando `for`

- Consiste da palavra chave `for` seguida de três expressões separadas por `ponto-e-vírgula` e entre parênteses
- Cada uma das três expressões é opcional e pode ser omitida
- No entanto, os dois `ponto-e-vírgulas` separando as expressões e os parênteses são `obrigatórios`

Comando `for`

- A construção com o `for` é equivalente ao uso do `while`, com a ordem de avaliação das expressões ilustrada a seguir

```
expr_inicial;  
while (expr_booleana) {  
    bloco de comandos  
    ...  
    expr_de_incremento  
}
```

Comando `for`

- A *`expr_inicial`* é avaliada uma única vez antes da execução do laço
- Em seguida, a expressão *`booleana`*, que controla a execução do laço, é avaliada e, enquanto for verdadeira, o bloco de comandos é executado
- Imediatamente após cada execução do bloco de comandos, a expressão *`booleana`* volta a ser avaliada

Comando `for`

- O comando `for` é executado da seguinte forma
- se existir, `expr_inicial` é avaliada
- se existir, `expr_booleana` é avaliada
 - se o resultado for zero a execução do `for` está completa
 - caso contrário, vai para o passo 3
- o corpo do `for` é executado
- se existir, `expr_de_incremento` é avaliada
- volta ao passo 2

Comando `for`

- A execução do comando `for` termina quando `expr_booleana` é avaliada em falso (zero) ou quando o controle é transferido para fora do bloco do `for` por um comando de `return`, `goto` ou `break`
- A execução do comando `continue` dentro do bloco do `for` possui o efeito de pular para o passo 4

Comando for

- Ilustração da utilização do comando **for** para cálculo do fatorial

```
/* Fatorial (versao 2) */
#include <stdio.h>
int main (void) {
    int i, n, f = 1;
    printf("Digite um número inteiro nao negativo:");
    scanf("%d", &n);
    /* calcula fatorial */
    for (i = 1; i <= n; i = i + 1){
        f = f * i;
    }
    printf(" Fatorial = %d \n", f);
    return 0;
}
```

Construções com laços

- Observe que as chaves que seguem o comando **for**, neste caso, são desnecessárias, já que o corpo do bloco é composto por um único comando
- Tanto a construção com **while** como a construção com **for** avaliam a expressão **booleana** que caracteriza o teste de encerramento no início do laço
- Assim, se esta expressão tiver valor igual a zero (falso), quando for avaliada pela primeira vez, os comandos do corpo do bloco não serão executados nem uma vez

Operador vírgula

- O operador vírgula pode ser útil no comando **for**
- Ele permite múltiplas inicializações e múltiplas atualizações
- Por exemplo, o comando

```
for (soma = 0, i = 1; i <= n; i = i + 1)
    soma = soma + i;
```

– pode ser usado para calcular a soma dos inteiros de 1 a n

Operador vírgula

- Seguindo o mesmo raciocínio pode-se colocar o corpo do laço dentro dos parênteses do **for**

```
for (soma = 0, i = 1; i <= n; soma = soma + i,  
    i = i + 1);
```

– mas não

```
for (soma = 0, i = 1; i <= n; i = i + 1, soma =  
    soma + i);
```

– na expressão **i = i + 1, soma = soma + i**

– a expressão **i = i + 1** é avaliada primeiro, conseqüentemente **soma** terá um valor diferente do desejado

Instrução repita-até

- O formato é

repita <bloco> **até que** <condição>

- <bloco> é executado pelo menos uma vez porque somente após a sua execução a <condição> é testada
- dentro do loop deve existir uma instrução que altera o valor da <condição>

Comando do-while

- C provê outro comando para construção de laços cujo teste de encerramento é avaliado no final
- Esta construção é o **do-while**, cuja forma geral é

```
do{  
    bloco de comandos  
} while (expr_booleana);
```

Comando do-while

- A execução do comando **do-while** termina quando *expr_booleana* é avaliada em falso (zero) ou quando o controle é transferido para fora do bloco do **do-while** por um comando de **return**, **goto** ou **break**
- A execução do comando **continue** dentro do bloco do **do-while** pode alterar a execução do comando **do-while**

Comando do-while

- Um exemplo do uso desta construção é mostrado a seguir
 - para cálculo do fatorial
 - em que validamos a inserção do usuário
 - isto é, o programa repetidamente requisita a inserção de um número enquanto o usuário inserir um inteiro negativo
 - cujo fatorial não está definido

Comando do-while

```
/* Fatorial (versao 3) */
#include <stdio.h>
int main (void) {
    int i, n, f = 1;
    /* requisita valor do usuário */
    do {
        printf("Digite um valor inteiro nao negativo:");
        scanf ("%d", &n);
    } while ( n<0 );
    /* calcula fatorial */
    for (i = 1; i <= n; i++)
        f *= i;
    printf(" Fatorial = %d\n", f);
    return 0;
}
```

Interrupções com **break** e **continue**

- A linguagem **C** oferece ainda duas formas para a interrupção antecipada de um determinado laço
- O comando **break**, quando utilizado dentro de um laço, interrompe e termina a execução do mesmo
- A execução prossegue com os comandos subsequentes ao bloco

Interrupções com `break` e `continue`

- O código abaixo ilustra o efeito de sua utilização

```
#include <stdio.h>
int main (void) {
    int i;
    for (i = 0; i < 10; i++){
        if (i == 5)
            break;
        printf("%d ", i);
    }
    printf("fim\n");
    return 0;
}
```

- A saída deste programa, se executado, será

0 1 2 3 4 fim

- pois, quando `i` tiver o valor `5`, o laço será interrompido e finalizado pelo comando `break`, passando o controle para o próximo comando após o laço, no caso uma chamada final de `printf`

Interrupções com `break` e `continue`

- O comando `continue` também interrompe a execução dos comandos de um laço
- A diferença básica em relação ao comando `break` é que o laço não é automaticamente finalizado
- O comando `continue` interrompe a execução de um laço passando para a próxima iteração
 - pulando qualquer código intermediário
 - para o comando `for`, `continue` faz com que a expressão booleana e a expressão de incremento sejam executados

Interrupções com `break` e `continue`

```
#include <stdio.h>
int main (void) {
    int i;
    for(i = 0; i < 10; i++) {
        if (i == 5) continue;
        printf("%d ", i);
    }
    printf("fim\n");
    return 0;
}
```

gera a saída:

```
0 1 2 3 4 6 7 8 9 fim
```

Interrupções com `break` e `continue`

```
/* INCORRETO */
#include <stdio.h>
int main (void) {
    int i = 0;
    while (i < 10) {
        if (i == 5) continue;
        printf("%d ", i);
        i++;
    }
    printf("fim\n");
    return 0;
}
```

- Deve-se ter cuidado com a utilização do comando `continue` nos laços `while`
- O programa ao lado é um programa **INCORRETO**, pois o laço criado não tem fim
 - a execução do programa não termina
 - isto porque a variável `i` nunca terá valor superior a `5`, e o teste será sempre verdadeiro
 - o que ocorre é que o comando `continue` "pula" os demais comandos do laço quando `i` vale `5`, inclusive o comando que incrementa a variável `i`

Exemplos

- Em **C**, existem duas formas usuais de se escrever um laço que “nunca termina” (as vezes chamado de “do forever” loop)

```
for (;;) {  
    bloco de comandos  
}
```

```
while (1) {  
    bloco de comandos  
}
```

- Os laços podem ser finalizados por um comando de **goto**, **break**, **return** dentro do laço

Exemplos

```
...
ch = '\0';
for ( ; ; ){
    ch = getchar(); /* obtem um caractere */
    if (ch== 'A') break; /* sai do laco */
}
printf("Voce digitou um A");
```

- ...
- Esse laço será executado até que o usuário digite um **A**

Comando `switch`

- Além da construção `else-if`, C provê um comando (`switch`) para selecionar um dentre um conjunto de possíveis casos
- Sua forma geral é

Comando switch

```
switch ( expr ){
  case op1:
    ... /* comandos ejecutados se expr == op1 */
  break;
  case op2:
    ... /* comandos ejecutados se expr == op2 */
  break;
  case op3:
    ... /* comandos ejecutados se expr == op3 */
  break;
  default:
    ... /* ejecutados se expr for diferente de
  todos */
  break;
}
```

Comando `switch`

- `opi` deve ser um número inteiro ou uma constante caractere
- Se `expr` resultar no valor `opi`, os comandos que se seguem ao caso `opi` são executados, até que se encontre um `break`
- Se o comando `break` for omitido, a execução do caso continua com os comandos do caso seguinte
- Se `expr` resultar em um valor diferente de `opi`, mas existir um caso `default`, os comandos que se seguem ao caso `default` são executados, até que se encontre um `break`

Comando `switch`

- Se `expr` resultar em um valor diferente de `opi`, e não existir um caso `default`, nenhum comando é executado, o controle é transferido para o próximo comando depois do `switch`
- O caso `default` (nenhum dos outros) pode aparecer em qualquer posição, mas normalmente é colocado por último
- Para exemplificar, mostramos a seguir um programa que implementa uma calculadora convencional que efetua as quatro operações básicas

Comando switch - exemplo

```
/* calculadora de quatro operações */  
#include <stdio.h>  
int main (void){  
    float num1, num2;  
    char op;  
    printf("Digite: numero op numero\n");  
    scanf ("%f %c %f", &num1, &op, &num2);  
    switch (op){  
        case '+':  
            printf(" = %f\n", num1+num2);  
            break;  
        case '-':  
            printf(" = %f\n", num1-num2);  
            break;  
        case '*':  
            printf(" = %f\n", num1*num2);  
            break;  
        case '/':  
            printf(" = %f\n", num1/num2);  
            break;  
        default:  
            printf("Operador invalido!\n");  
            break;  
    }  
    return 0;  
}
```