Linguagem C - Notas de Aula

Profª. Carmem Hara e Prof. Wagner Zola
Revisão: Prof. Armando Luiz N. Delgado

Agosto 2003

1 Programas C

Um programa C consiste de uma ou mais partes chamadas funções. Um programa em C consiste de pelo menos uma função chamada **main**. Esta função marca o ponto de início de execução do programa.

Programas C tem a seguinte estrutura geral:

```
#include <stdio.h>

definição de constantes

funções

main()
{
        declaração de variáveis
        ....
        sentenças
        ....
}
```

1.1 Senteças: simples e compostas

Cada instrução em C é chamada de sentença. Sentenças simples são terminadas com um ponto e vírgula. Usando chaves, podemos agrupar sentenças em blocos, chamados de sentenças compostas.

Exemplos de sentenças incluem:

- Simples:

```
x = 3;
```

- Composta:

```
{
   i = 3;

   printf("%d\n", i);

   i = i + 1;
}
```

O corpo da função main() é um exemplo de sentença composta.

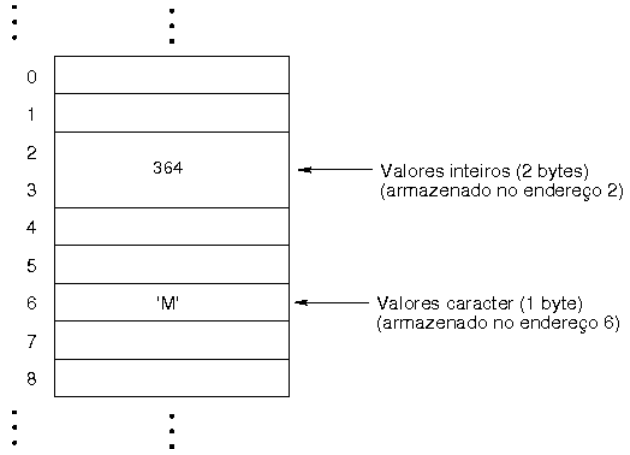
1.2 Variáveis em C

Uma variável é uma informação qe você pode usar dentro de um programa C . Esta informação está associada com um lugar específico da memória (isso é feito pelo compilador). O **nome** da variável e o endereço da memória onde a informação está armazenada estão associados. O nome e o endereço não mudam. Mas, o valor da informação pode mudar (o valor do que está dentro da caixa pode mudar, embora o tipo seja sempre o mesmo). Cada variável tem um tipo associado. Alguns tipos de variáveis que discutiremos incluem int, char e float.

Cada variável usa uma determinada quantidade de armazenamento em memória. A maneira como sabemos quantos bytes são utilizados é pelo tipo da variável. Variáveis do mesmo tipo utilizam o mesmo número de bytes, não interessando qual o valor que a variável armazena.

Um dos tipos utilizados para armazanar números é o int. Ele é usado para armazenar números inteiros. Em Borland C++ , valores do tipo int usam 2 bytes de memória e podem armazenar valores de -32768 a 32767. Por exemplo, 42, 1492, ou -3691.

Outro tipo é o char, usado para armazenar caracteres. Um caracter é um símbolo (uma letra do alfabeto, um dígito, um símbolo de pontuação, etc). Um char é armazenado em 1 byte de memória. Cada caracter é associado com um valor entre 0 e 255. O compilador C faz a tradução para você, portanto você não precisa saber estes números. Em C , um caracter é representado entre apóstrofes (’). Por exemplo, ’C’, ’a’, ’5’, ’$’. Note que ’5’ é um caracter, e não o inteiro 5.



A figura acima mostra como um int e um char são armazenados na memória.

Outro tipo existente é o float, usado para armazenar números reais (números com o ponto decimal). Este números são armazenados em duas partes: a *mantissa* e o *expoente*. Eles são armazenados de uma maneira que se assemelha a notação exponencial. Por exemplo, o número 6.023×10^{23} é escrito como $6.023e23$. Neste caso, a mantissa é 6.023 e o expoente 23.

Estes números são armazenados de uma forma padrão, tal que a mantissa tem apenas um dígito para a esquerda do ponto decimal. Desta forma, 3634.1 é escrito como 3.6341e3, e 0.0000341 é escrito 3.41e-5. Note também que a precisão é limitada pela mantissa. Somente os 6 dígitos mais significativos são armazenados. Em Borland C++ um float ocupa 4 bytes de memória. Há muitos outros tipos (short, long, double), que serão descritos no futuro.

1.3 Definição de Variável em C

Se você usa variáveis no programa, você deve defini-las. Isto envolve especificar o tipo da variável e o seu nome. As regras para formar nomes de variáveis em C são:

- qualquer sequência de letras, digitos, e ’_’, MAS DEVE COMEÇAR com uma letra ou com ’_’. Por exemplo, hora_inicio, tempo, var1 são nomes de variáveis válidos, enquanto 3horas, total$ e azul-claro não são nomes válidos;
- Maiúsculas $\neq$ Minúsculas;
- Não são permitidos nomes ou palavras reservadas da linguagem.

É sempre uma boa idéia ter certas regras (para você mesmo) para nomear variáveis para tornar o programa mais legível:

- Dê nomes significativos as variáveis (mas não muito longos);
- Use nomes de variáveis do tipo i, j, k somente para variáveis tipo contadores;

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

Tabela 1: Palavras Reservadas da Linguagem C

- Pode-se usar letras maiúsculas ou '_' para juntar palavras. Por exemplo, horaInicio ou hora_inicio. Use o que você preferir, mas SEJA **CONSISTENTE** em sua escolha.

Os tipos básicos de dados existentes em C são:

Tipo de Dado	Bits	Faixa de Valores
char	8	-128 a 127
int	16	-32768 a 32767
float	32	6 a 7 dígitos significativos
double	64	15 a 16 dígitos significativos

Abaixo está um exemplo de um programa com diversas definições de variáveis:

```
main()
{
    int pera;
    char qualidade;
    float peso;

    pera = 3;
    qualidade = 'A';
    peso = 0.653;
    ...
}
```

Quando variáveis são definidas, elas não possuem valores ainda. Nós damos valores as variáveis usando o *operador de atribuição* (=). Variáveis também podem ser inicializadas para conter valores quando são definidas. Usando esta forma, o program acima ficaria:

```
main()
{
    int pera = 3;
    char qualidade = 'A';
    float peso = 0.653;

    ...
}
```

Para resumir: quando um programa é executado, **uma variável** é associada com:

- **um tipo:** diz quantos bytes a variável ocupa, e como ela deve ser interpretada.

- **um nome:** um identificador.
- **um endereço:** o endereço do byte menos significativo do local da memória associado a variável.
- **um valor:** o conteúdo real dos bytes associados com a variável; o valor da variável depende do tipo da variável; a definição da variável não dá valor a variável; o valor é dado pelo operador de atribuição, ou usando a função scanf(). Nós veremos mais tarde que a função scanf() atribui a uma variável um valor digitado no teclado.
- Em C , nomes de variáveis devem ser declarados antes de serem usados. Se não for declarado, ocorrerá um erro de compilação.
- Devem ser dados valores às variáveis antes que sejam utilizadas. Se você tentar utilizar a variável antes de especificar o seu valor, você obterá "lixo" (o que quer que esteja armazenado no endereço da variável na memória quando o programa começa sua execução), culminando com falha na execução do programa.

1.4 Constantes

Em C , além de variáveis, nós podemos usar também números ou caracteres cujos valores não mudam. Eles são chamados de constantes. Constantes não são associados a lugares na memória.

Assim como variáveis, constantes também têm tipos. Uma constante pode ser do tipo int, char, etc. Você nao tem que declarar constantes, e pode utilizá-las diretamente (o compilador reconhece o tipo pela maneira que são escritos). Por exemplo, 2 é do tipo int, e 2.0 é do tipo double. Por convenção, todas as constantes reais são do tipo double.

1.5 Caracteres Constantes

Um constante caracter é escrita entre apóstrofes, como em 'A'. Todas as letras, números e símbolos que podem ser impressos são escritos desta forma em C . Às vezes precisamos de caracteres que não podem ser impressos, por exemplo, o caracter de "nova linha", que não tem uma tecla específica no teclado. Neste caso, usa-se *caracteres de escape*. Tais caracteres são escritos não somente como um símbolo entre apóstrofes, mas como um sequência de caracteres entre apóstrofes. Por exemplo, `'\n'` é o caracter para nova linha (uma sequência que inicia com a barra invertida é chamada de *sequência de escape*). Se quisermos representar o caracter de barra invertida, temos que escrever `'\\'`. Note que `\n` é o caracter de nova linha - embora use-se dois símbolos para representá-lo. A barra invertida é chamada de *escape*. Ele diz ao compilador que o n que segue não é a letra n, mas que a sequência completa de caracteres deve ser interpretada como o caracter de "nova linha".

Cada caracter constante tem um valor inteiro igual ao seu valor numérico do seu código ASCII. Por exemplo, considere a constante 'A', que tem código ASCII 65, e 'B' que tem código 66. Nós podemos usar a expressão 'A' + 1. O resultado é o valor 66. E se o tipo da expressão resultante for char, então o resultado da expressão é 'B'.

1.6 Entrada e Saída

Se quisermos que um programa C mostre alguns resultados, or se quisermos que o programa peça ao usuário que entre com alguma informação, nós podemos usar as funções existentes em C chamadas printf() e scanf(). Se você quiser usar estas funções em seu programa, voce deve incluir a seguinte linha no início do seu programa:

#include <stdio.h>

Isto causa que o arquivo *header* chamado stdio.h seja incluído no seu arquivo fonte. Este arquivo contém *protótipos* das funções print() e scanf(). Ele declara ao compilador o nome das funções e algumas informações adicionais necessárias para que as instruções sejam executadas corretamente.

1.7 Exibindo informações na tela: printf()

printf() pode ser utilizado para imprimir mensagens e valores em uma variedade de formatos. Por enquanto, printf() é melhor descrito através de exemplos.

```
printf("Alo todo mundo");
```

Imprimirá Alo todo mundo para o usuário.

Para dizer a função printf exatamente o que fazer, nós devemos especificar o que será impresso. Nós devemos dar a função o que chamamos de *argumentos*. No exemplo acima, "Alo todo mundo" é um argumento para a função printf().

O primeiro argumento do printf() é sempre um *string* (uma série de caracteres entre aspas (")).

Nós também podemos colocar caracteres de escape no string para imprimir caracteres especiais. Por exemplo, colocando `\n` no string causa que o restante do string seja impresso na linha seguinte. Outros caracteres de escape serão apresentados no futuro.

Se quisermos imprimir o valor de expressões variáveis, argumentos adicionais são necessários. Dizemos ao printf() como mostrar valores de expressões usando *especificadores de formato*. Podemos colocar %c, %d, %f (ou outros especificadores de formato listados no texto) dentro do primeiro argumento para especificar o que queremos dar display. Nós então passamos argumentos adicionais que ser referem aos especificadores de formato (na ordem em que eles ocorrem). Este argumentos podem ser constantes ou variáveis, ou alguma expressão mais complicada. O que quer que eles sejam, eles devem ser avaliados e os valores obtidos e impressos de acordo com os especificadores de formato. Considere o seguinte programa:

```
#include <stdio.h>
#define PRECO 1.99

main()
{
        int pera = 3;
        char qualidade = 'A';
        float peso = 2.5;

        printf("Existem %d peras de qualidade %c ", pera, qualidade);
        printf("pesando %f quilos.\n", peso);
        printf("O preco por quilo e %f, total e %f\n", PRECO, peso * PRECO);
}
```

A saída do programa será:

```
Existem 3 peras de qualidade A pesando 2.500000 quilos.
O preco por quilo e 1.990000, total e 4.975000
```

A linha #define PRECO 1.99 no início do programa define uma *macro*. Ou seja, definimos que PRECO é um sinônimo para 1.99 e, portanto, toda ocorrência de PRECO no programa é substituído por 1.99 antes que ele seja compilado.

Nós também podemos especificar o tamanho utilizado para impressão da seguinte forma:

%6d inteiro, com pelo tamanho pelo menos 6
%6f ponto flutuante, com tamanho pelo menos 6
%.3f ponto flutuante, com 3 digitos depois do ponto decimal
%6.3fponto flutuante, com tamanho pelo menos 6 e 3 digitos depois do ponto decimal
%6.0fponto flutuante, com pelo menos tamanho 6 e nenhum digito depois do ponto decimal.

Note que a especificação de tamanho simplesmente determina o tamanho mínimo. Se o número não couber no tamanho especificado, o número completo será mostrado.

Quando utilizar a função printf() tenha cuidado para especificar o tipo correto dos argumentos. Se o tipo do argumento não for correto, o compilador Borland C++ não acusará erro, e um valor incorreto será mostrado. Por exemplo, no programa abaixo que está incorreto:

```
#include <stdio.h>

main()
{
    printf("Exemplo errado: %d\n", 3.14159);
}
```

O resultado do programa será alguma coisa como:

```
Exemplo errado: -31147
```

1.8 Lendo informação: scanf()

scanf() pode ser usado para ler valores digitados no teclado. Estes valores são lidos de acordo com *especificadores de conversão*, que são especificados pelo programador como argumentos do scanf().

Considere o seguinte programa:

```
#include <stdio.h>

main()
{
    int idade;

    printf("Entre sua idade: ");
    scanf("%d", &idade);

    printf("Voce tem %d anos\n", idade);
}
```

Este programa mostrará no monitor: Entre sua idade: e aguardará que um número seja digitado e a tecla ENTER. Depois disso, a variável idade conterá o valor digitado pelo usuário.

Assim como com o printf(), o primeiro argumento é o especificador de formato. Os próximos argumentos correspondem a o que está sendo especificado pelo primeiro argumento.

Note o & precedendo a variável idade. Simplesmente lembre-se que você geralmente precisará colocar um & precedendo nomes de variáveis em um scanf(). Você sempre precisará usá-lo antes de variáveis do tipo primário como os discutidos até este momento (int, char, float, e suas versões long e unsigned).

Mais de um valor pode ser lido por um mesmo scanf(). Considere o seguinte exemplo:

```
#include <stdio.h>

main()
{
    int dia, mes, ano;

    printf("Entre com a data do seu aniversario (dd mm aa): ");
    scanf("%d %d %d", &dia, &mes, &ano);
```

```
        printf("Voce nasceu em %d/%d/%d\n", dia, mes, ano);
    }
```

Este exemplo funciona exatamente como o exemplo anterior. Um único scanf() lê os 3 números quando estes números são separados por espaços (espaços em branco, tabulação, novas linhas). Então você pode teclar ENTER depois de cada número, ou colocar espaços ou tabulações entre os números. Os espaços são ignorados pelo scanf(). Os brancos na especificação de formato do scanf(), "%d %d %d" são simplesmente para facilitar a leitura do programa, e não tem nada a ver com os espaços ignorados pelo scanf(). Se tivéssemos escrito "%d%d%d", o scanf() funcionaria da mesma forma. Os espaços em branco simplesmente são necessários para saber quando termina um número e começa o outro.

Porém se o scanf() estiver lendo caracteres (%c), os espaços não são ignorados, já que espaços são caracteres válidos na linguagem. Por exemplo, o código ASCII para espaço em branco e' 32.

1.9 Algoritmo X Programa

```
ALGORITMO PERIMETRO_AREA

  /* Calcula o perímetro e a area de uma circunferencia
     de raio R (fornecido pelo usuario) */

  /* Definir variaveis */
        int Raio;
        float Perim, Area, PI;
        PI = 3.14159;

  /* Obter Raio da circunferencia */
        Escreva("Entre com o valor do raio:");
        Leia(Raio);

  /* Calcular Perimetro do Circulo */
        Perim = 2 * PI * Raio;

  /* Calcular Area da Circunferencia */
        Area = PI * Raio ** 2;

  /* Exibir Resultados */
        Escreva("O perimetro da circunferencia de raio", Raio, "eh", Perim);
        Escreva("e a area eh ",Area);

  /* Terminar Programa */

FIM_ALGORITMO PERIMETRO_AREA
```

Programa em C

```
  /* programa que calcula o perímetro e a área de uma
     circunferência de raio R (fornecido pelo usuário) */

#include <stdio.h>  /* inclui diretivas  de     entrada-saída */
#include <math.h>  /*  inclui diretivas das funções matemáticas */
```

```
#define  PI  3.14159

main()
{
     /* Definir variaveis */
        int Raio;
        float  Perim, Area;

     /* Obter Raio da circunferencia */
        printf("Entre com o valor do raio: ");
        scanf("%d", &Raio);

     /* Calcular Perimetro do Circulo */
        Perim = 2 * PI * Raio;

     /* Calcular Area da Circunferencia */
        Area  = PI * pow(Raio, 2);

     /* Exibir Resultados */
        printf("O perimetro da circunferencia de raio %d eh %.2f  \n", Raio, Per
        printf("e a area eh %.2f", Area);
}
```

2 Operações Aritméticas e Expressões. Operações Relacionais.

2.1 Operações Aritméticas

Em C, nós podemos executar operações aritméticas usando variáveis e constantes. Algumas operações mais comuns são:

+ adição

- subtração

***** multiplicação

/ divisão

% resto (módulo)

Estas operações podem ser usadas como mostram os exemplos abaixo, assumindo que as variáveis necessárias já estão declaradas:

```
celsius = (fahrenheit - 32) * 5.0 / 9.0;

forca =  massa * aceleracao;

i = i + 1;
```

2.1.1 Precedência de Operadores

Em C, assim como em álgebra, há uma ordem de precedência de operadores.

Assim, em $(2 + x)(3x^2 + 1)$, expressões em parêntesis são avaliadas primeiro, seguidos por exponenciação, multiplicação, divisão, adição e subtração.

Da mesma forma, em C, expressões entre parêntesis são executadas primeiro, seguidas de `*`, `/` and `%` (que tem todos a mesma precedência), seguido de + and − (ambos com a mesma precedência).

Quando operações adjacentes têm a mesma precedência, elas são associadas da esquerda para a direita. Assim, `a * b / c * d % e` é o mesmo que `(((a * b) / c) * d) % e)`.

2.1.2 A Operação de Resto (%)

Esta operação é usada quando queremos encontrar o resto da divisão de dois inteiros. Por exemplo, 22 dividido por 5 é 4, com resto 2 ($4 \times 5 + 2 = 22$).

Em C, a expressão `22 % 5` terá valor `2`.

Note que `%` só pode ser utilizados entre dois inteiros. Usando ele com um operando do tipo `float` causa um erro de compilação (como em `22.3 % 5`).

2.1.3 Expressões e Variáveis

Expressões aritméticas podem ser usadas na maior parte dos lugares em que uma variável pode ser usada.

O exemplo seguinte é válido:

```
int raio = 3 * 5 + 1;

printf("circunferencia = %f\n", 2 * 3.14 * raio);
```

Exemplos de lugares onde uma expressão aritmética NÃO pode ser usada incluem:

```
int yucky + 2 = 5;

scanf("%d", &(oops * 5))
```

Este exemplo é ilegal e causará erro de compilação.

2.2 Operação de Atribuição Aritmética

É freqüente em programas C expressões do tipo:

```
tudo  = tudo + parte;

tamanho = tamanho * 2.5;

x = x * (y + 1);

j = j - 1;
```

C fornece operadores adicionais que podem ser usados para tornar estes tipos de atribuições mais curtos. Há um operadorador de atribuição para cada operação aritmética listada anteriormente:

+= operação de atribuição de adição

-= operação de atribuição de subtração

***=** operação de atribuição de multiplicação

/= operação de atribuição de divisão

%= operação de atribuição de resto

Cada uma dessas operações podem ser usadas para tornar as expressões anteriores mais curtas:

```
tudo += parte;

tamanho *= 2.5;

x *= y + 1;

j -= 1;
```

2.2.1 Operador de Incremento

Há alguns operadores em C que são equivalentes as seguintes expressões (que são bastante comuns em programas):

```
k = k + 1;

j = j - 1;
```

Estes operadores adicionais, que são ++ and −−, podem ser usados para encurtar as operações acima:

```
k++;

j--;
```

Estes operadores também podem ser colocados depois do nome da variável:

```
++k;

--j;
```

O fato do operador de incremento ser colocado antes ou depois da variável não altera o efeito da operação – o valor da variável é incrementada ou decrementada de um. A diferença entre os dois casos é QUANDO a variável é incrementada. Na expressão `k++`, o valor de `k` é primeiro usado e então é incrementada – isto é chamado *pós-incremento*. Na expressão `++k`, `k` é incrementado primeiro, e então o valor (o novo valor) de `k` é usado – isso é chamado *pré-incremento*.

A diferença é ilustrada nos seguintes exemplos:

```
main()
{
   int k = 5;

   printf("k = %d\n", k);
   printf("k = %d\n", k++);
   printf("k = %d\n", k);
}
```

O programa acima (que usa pós-incremento) imprimirá o seguinte:

```
k = 5
k = 5
k = 6
```

A segunda linha impressa com o valor de `k` é `5` porque o valor de `k++` era `5`, e `k` é `6` depois da impressão.

Para o programa:

```
main()
{
   int k = 5;

   printf("k = %d\n", k);
   printf("k = %d\n", ++k);
   printf("k = %d\n", k);
}
```

O programa, que usa pré-incremento, terá a seguinte saída:

```
k = 5
k = 6
k = 6
```

A segunda linha impressa é `6` porque o valor de `++k` é `6`.

Os operadores de atribuição não podem ser usados com expressões aritméticas. Por exemplo, as expressões

```
(ack + 2)++;

(nope + 3) += 5;
```

resultarão em erros de compilação.

Finalmente, quando usar o operador de incremento em um `printf()`, tome cuidado para não fazer o seguinte:

```
printf("%d %d\n", ++uhoh, uhoh * 2);
```

Embora isso seja perfeitamente legal em C, os resultados não são garantidados que sejam consistentes. A razão para isso é que não há garantia que os argumentos do `printf()` sejam avaliados em uma determinada ordem. O resultado do `printf()` será diferente dependendo se `++uhoh` é avaliado primeiro ou depois de `uhoh * 2`.

A solução para este problema é escrever o seguinte:

```
++uhoh;
printf("%d  %d\n", uhoh, uhoh * 2);
```

2.3 Operadores Relacionais

Em C, há operadores que podem ser usados para comparar expressões: os operadores relacionais.

Há seis operadores relacionais em C:

$<$ menor que

$>$ maior que

$<=$ menor ou igual que ($\leq$)

$>=$ maior ou igual que ($\geq$)

== igual a

!= não igual a ($\neq$)

Os resultados deste operadores é `0` (correspondendo a *falso*), ou `1` (correspondendo a *verdadeiro*). Valores como esses são chamados valores *booleanos*. Algumas linguagens de programação como Pascal tem um tipo de variável distinto para valores booleanos. Este não é o caso do C, onde valores booleanos são armazenados como variáveis numéricas tais como o `int`.

Considere o seguinte programa:

```
main()
{
    int idade;

    idade = 17;
    printf("Pode tirar carteira de motorista? %d\n", idade >= 18);
    idade = 35;
    printf("Pode tirar carteira de motorista? %d\n", idade >= 18);
}
```

A saída deste programa será:

```
Pode tirar carteira de motorista? 0
Pode tirar carteira de motorista? 1
```

Na primeira linha, `idade` é `17`. Logo, `17 >= 18` é falso, que é `0`.

Depois disso, `idade` é `35`. Logo, `35 >= 18` é verdadeiro, que é `1`.

Note também que o operador de igualdade é escrito com "sinais de igual duplo", ==, não =. Tenha cuidado com esta diferença, já que colocar = no lugar de == não é um erro sintático (não gera erro de compilação), e não significa o que você espera.

2.3.1 Precedência dos operadores relacionais

Operadores aritméticos tem precedência maior que os operadores relacionais. Por exemplo, a expressão `3 + 5 < 6 * 2` é o mesmo que `(3 + 5) < (6 * 2)`.

Se por alguma razão você quer que o resultado do uma operação relacional em uma expressão aritmética, é necessário usar parêntesis. Por exemplo, a expressão `score + (score == 0)` será sempre igual ao valor de `score`, exceto quando o valor de `score` seja `0`. Neste caso, o valor da expressão é `1` (porque `(score == 0)` é igual a 1).

Uma observação sobre valores booleanos – embora você possa assumir que o valor de uma operação relacional é `0` ou `1` em C, **qualquer valor diferente de zero é considerado verdadeiro**. Falaremos sobre isso mais tarde durante o curso.

2.4 Revisão de Expressões:

O que é impresso pelos seguintes programas:

```
#include <stdio.h>

main() {
  int score = 5;

  printf(``%d'', 5 + 10 * 5 % 6);        ==> 7
  printf(``%d'', 10 / 4);                ==> 2
  printf(``%f'', 10.0 / 4.0);            ==> 2.5
  printf(``%c'', 'A' + 1);               ==> B
  printf(``%d'', score + (score == 0));  ==> 5
}
```

```
#include <stdio.h>

main() {
  int n1, n2, n3;

  printf(``Entre com um numero inteiro: ``);
  scanf(``%d'', &n1);
  n1 += n1 * 10;
  n2 = n1 / 5;
  n3 = n2 % 5 * 7;
  n2 *= n3-- % 4;
  printf(``%d %d %d'', n2, n3, n2 != n3 + 21);
}
```

Como é a seguinte expressão completamente parentizada ?

```
a * b / c + 30 >= 45 + d * 3 ++e == 10
```

2.5 Exemplo de programas

Exemplo 1: escreva um programa que leia um número inteiro e imprima 0 se o número for par e 1 se o número for ímpar.

```
#include <stdio.h>

main() {
  int numero;

  printf(``Entre com um numero inteiro: ``);
  scanf(``%d'', &numero);
  printf(``\nPar? %d\n'', numero % 2 );
}
```

Exemplo 2: escreva um programa que leia 3 números inteiros e calcule a soma, média, e produto.

```
#include <stdio.h>

main() {
    int n1, n2, n3;
    int soma;

    printf( "Entre com 3 numeros inteiros: ");
    scanf( "%d %d %d",&n1, &n2, &n3);
    soma = n1 + n2 + n3;
    printf( "Soma = %d\n", soma );
    printf( "Media = %8.2f\n", soma / 3.0 );
    printf( "Produto = %d\n", n1 * n2 * n3 );
}
```

2.6 Precedência e associatividade de operadores

Operador	Associatividade
`()`	esquerda para direita
`++ -- &` (unários)	direita para esquerda
`* / %`	esquerda para direita
`+ -`	esquerda para direita
`< <= > >=`	esquerda para direita
`== !=`	esquerda para direita
`= += -= *= /= %=`	direita para esquerda

3 Expressões como valores

Em C, todas as expressões são avaliadas. O resultado da avaliação é um valor e pode ser usado em quaisquer lugares.

3.1 Expressões aritméticas, relacionais e lógicas

Como você já sabe, expressões usando operadores aritméticos, relacionais e lógicos[1] são avaliados. O valor resultante é um número. Para os operadores relacionais e lógicos, este número pode ser 0 (que significa falso) ou 1 (que significa verdadeiro). Por exemplo:

`3 + 5 * 4 % (2 + 8)`	tem valor 3;
`3 < 5`	tem valor 1;
`x + 1`	tem valor igual ao valor da variável `x` mais um;
`(x < 1) || (x > 4)`	tem valor 1 quando o valor da variável `x` é fora do intervalo [1,4], e 0 quando `x` está dentro do intervalo.

3.2 Expressões envolvendo o operador de atribuição (=)

O formato do operador de atribuição é:

$$lvalue = expressao \tag{1}$$

Um $lvalue$ (do inglês "left-hand-side value" - valor a esquerda) é um valor que se refere a um endereço na memória do computador. Até agora, o único "lvalue" válido visto no curso é o nome de uma variável. A maneira que a atribuição funciona é a seguinte: a expressão do lado direito é avaliada, e o valor é copiado para o endereço da memória associada ao "lvalue". O tipo do objeto do "lvalue" determina como o valor da $expressao$ é armazenada na memória.

Expressões de atribuição, assim como expressões, têm valor. O valor de uma expressão de atribuição é dado pelo valor da expressão do lado direito do =. Por exemplo:

`x = 3`	tem valor `3`;
`x = y+1`	tem o valor da expressão `y+1`.

Como consequência do fato que atribuições serem expressões que são associadas da direita para esquerda, podemos escrever sentenças como:

```
i = j = k = 0;
```

Que, usando parênteses, é equivalente a `i = (j = (k = 0))`. Ou seja, primeiro o valor 0 é atribuído a `k`, o valor de `k = 0` (que é zero) é atribuído a `j` e o valor de `j = (k = 0)` (que também é zero) é atribuído a `i`.

Uma característica muito peculiar de C é que expressões de atribuição podem ser usados em qualquer lugar que um valor pode ser usado. Porém você deve saber que usá-lo dentro de outros comandos produz um efeito colateral que é alterar o valor da variável na memória. Portanto, a execução de:

```
int quadrado, n = 2;

printf("Quadrado de %d eh menor que 50? %d \n", n, (quadrado = n * n) < 50);
```

[1] Operadores lógicos `&&` e `||` serão vistos na próxima aula.

causa não apenas que o valor `4` seja impresso, como a avaliação da expressão relacional dentro do printf() faz com que o número `4` seja copiado para o endereço de memória associado com a variável `quadrado`. Note que é necessário usar parênteses em `quadrado = n * n` já que = tem menor precedência que o operador relacional <.

Agora compare o exemplo anterior com o próximo, no qual o valor `4` é impresso, mas sem nenhum efeito colateral:

```
int quadrado, n = 2;

printf("Quadrado de %d eh menor que 50? %d \n", n, n * n < 50);
```

Note que agora não há necessidade de parênteses para a expressão `n * n` porque `*` tem maior precedência que o operador relacional <.

3.2.1 Operadores de atribuição aritmética

Como foi discutido em classe, estes comandos de atribuição funcionam de forma similar que o comando de atribuição. O lado esquerdo da expressão deve ser um *lvalue*. O valor da expressão de atribuição aritmética é igual ao valor da sentença de atribuição correspondente. Por exemplo:

`x += 3` é igual a `x = x + 3` e tem valor `x + 3`
`x *= y + 1` é igual a `x = x * (y + 1)` e tem valor `x * (y + 1)`

3.2.2 Operadores de incremento e decremento

Já que incremento e decremento são formas de atribuição, o operando deve ser um *lvalue*. O valor de uma expressão de incremento ou decremento depende se o operador é usado na notação PRé ou PóS fixada (`x++, ++x, x--, --x`). Se for pré-fixada, o valor da expressão é o novo valor após o incremento ou decremento. Se for pós-fixada, o valor da expressão é o valor antigo (antes do incremento ou decremento). Por exemplo no caso de incremento, a expressão:

`x++` tem o valor de `x`
`++x` tem o valor de `x + 1`

Note que não importando a notação usada, o valor de `x` (o conteúdo do endereço de memória associada a `x`) será `x + 1`. A diferença está no valor das expressões `x++` e `++x`, não no valor de `x` (em ambos os casos o valor de `x` será incrementada de um).

3.2.3 Ambiguidade em certas expressões

Às vezes, problemas podem acontecer devido o fato que C não especifica a ordem de avaliação dos operadores em uma operação binária. Em outras palavras, em expressões como `a + b` ou `a < b`, não há maneira de saber se o valor de `a` será avaliado antes ou depois de `b` (pense em `a` e `b` como sendo qualquer expressão, não somente variáveis.) Qual deles será avaliado primeiro é particular de cada compilador, e diferentes compiladores em máquinas diferentes podem ter resultados diferentes. Portanto, se a avaliação de um dos operadores pode alterar o valor do outro, o resultado pode ser diferente dependendo da ordem de avaliação. Portanto, em expressões do tipo `x + x++`, o valor pode diferir dependendo do compilador utilizado. Isto porque não sabemos quando exatamente o incremento de `x` ocorre. Outros maus exemplos: `y = x + x--` e `x = x++`. De forma geral, para evitar este problema, não utilize senteças como estas.

4 Mais sobre tipos: conversão implícita e explícita

Expressões não tem somente um valor, mas também tem um tipo associado.

Se ambos os operandos de uma operação aritmética binária são do mesmo tipo, o resultado terá o mesmo tipo. Por exemplo:

```
3 + 5       é 8, e o tipo é int
3.5 + 2.25 é 5.75, e o tipo é double
```

O único comportamento não óbvio é a da divisão de inteiros:

```
30 / 5 é 6
31 / 5 é 6
29 / 5 é 5
3 / 5  é 0
```

Lembre-se de evitar escrever algo como `1 / 2 * x` significando $\frac{1}{2}x$. Você sempre obterá o valor `0` porque `1 / 2 * x` é `(1 / 2) * x` que é `0 * x` que é `0`. Para obter o resultado desejado, você poderia escrever `1.0 / 2.0 * x`.

4.1 Conversão de tipos

Valores podem ser convertidos de um tipo para outro implicitamente, da forma já comentada nas aulas feito pelo compilador, ou explicitamente, usando um operador chamado *type casting*.

Em expressões envolvendo operadores binários com operandos de tipos diferentes, os valores dos operandos são convertidos para o mesmo tipo antes da operação ser executada: tipos mais simples são "promovidos" para tipos mais complexos. Portanto, o resultado da avaliação de uma expressão com operandos de tipos diferentes será o tipo do operando mais complexo. Os tipos em C são (do mais simples para o mais complexo):

```
char < int < long < float < double
```

O sinal de < significa que o tipo da esquerda é promovido para o tipo da direita, e o resultado será do tipo mais a direita. Por exemplo:

```
3.5 + 1 é 4.5
4 * 2.5 é 10.0
```

Esta regra estende-se para expressões envolvendo múltiplos operadores, mas você deve se lembrar que a precedência e associatividade dos operadores pode influenciar no resultado. Vejamos o exemplo abaixo:

```
main()
{
  int a, b;

  printf("Entre uma fracao (numerador e denominador): ")
  scanf("%d %d", &a, &b);

  printf("A fracao em decimal e  %f\n", 1.0 * a / b);
}
```

Multiplicando por `1.0` assegura que o resultado da multiplicação de `1.0` por `a` será do tipo real, e portanto, a regra de conversão automática evitará que o resultado da divisão seja truncado. Note que se tivéssemos primeiro feito a divisão `a/b` e depois multiplicado por `1.0`, embora o tipo da expressão `a/b*1.0` seja do tipo `double`, o valor da expressão seria diferente do valor de `1.0 * a/b`. Por que ?

Em atribuições, o valor da expressão do lado direito é convertido para o tipo da variável do lado esquerdo da atribuição. Isto pode causar promoção ou "rebaixamento" de tipo. O "rebaixamento" pode causar perda de precisão ou mesmo resultar em valores errados.

Em operações de atribuição, atribuir um `int` em um `float` causará a conversão apropriada, e atribuir um `float` em um `int` causará truncamento. Por exemplo:

```
float a = 3;     é equivalente a a = 3.0
int a = 3.1415; é equivalente a a = 3 (truncado)
```

Basicamente, se o valor da expressão do lado direito da atribuição é de um tipo que não cabe no tamanho do tipo da variável do lado esquerdo, resultados errados e não esperados podem ocorrer.

4.2 Modificadores de tipos

Os tipos de dados básicos em C podem estar acompanhados por modificadores na declaração de variáveis. Tais modificadores são: **long**, **short**, **signed** e **unsigned**. Os dois primeiros têm impacto no tamanho (número de bits) usados para representar um valor e os dois últimos indicam se o tipo será usado para representar valores negativos e positivos (**signed**) ou sem este modificador) ou apenas positivos (**unsigned**).

A Tabela 2 mostra uma lista completa de todos os tipos de dados em C, com e sem modificadores:

Modificador	Tamanho em bits	Faixa de valores
char	8	-127 a 127
unsigned char	8	0 a 255
int	16	-32767 a 32767
unsigned int	16	0 a 65535
short int	16	-32767 a 32767
unsigned short int	16	0 a 65535
long int	32	-2147483647 a 2147483647
unsigned long int	32	0 a 4294967295
float	32	Mantissa de 6 dígitos
double	64	Mantissa de 10 dígitos
long double	80	Mantissa de 10 dígitos

Tabela 2: Modificadores de Tipos de Dados

4.3 *Cast* de tipos

O C tem um operador para alterar o tipo de um valor explicitamente. Este operador é chamado de *cast*. Executando um *cast* de tipos, o valor da expressão é forçado a ser de um tipo particular, não importando a regra de conversão de tipos.

O formato do *cast* de tipos é:

$$(nome\text{-}do\text{-}tipo)\ expressao$$

O parênteses NÃO é opcional na expressão acima.
Podemos usar o *cast* de tipos da seguinte forma:

```
int fahr = 5;
float cels;

printf("Valor = %f\n", (float)fahr);
cels = (float)5 / 9 * (fahr - 32);
printf("celsius = %d\n", (int)cels);
```

Agora que conhecemos o operador de *cast* de tipo podemos reescrever o programa que faz a conversão de fração para decimal.

```
main()
{
  int a, b;

  printf("Entre com uma fracao (numerador e denominador): ")
  scanf("%d %d", &a, &b);

  printf("A fracao em decimal e  %f\n", (float) a / b);
}
```

O *cast* de tipo tem a maior precedência possível, portanto podemos fazer o *cast* de `a` ou de `b` para ser do tipo `float`, e não há necessidade de parênteses extra. No exemplo acima, o *cast* causa o valor da variável `a` ser convertido para `float`, mas não causa mudança no tipo da variável `a`. O tipo das variáveis é definido uma vez na declaração e não pode ser alterado.

5 Comandos de entrada e saída: getchar() e putchar()

Vamos discutir algumas funções de entrada de dados (diferente do scanf()). A entrada de texto é considerada como um *fluxo* de caratecteres. Um *fluxo texto* é uma sequência de caracteres dividida em linhas; cada linha consiste de zero ou mais caracteres seguido do caractere de nova linha (\n). Como programador, você não quer se preocupar em como as linhas são representadas fora do programa. Quem faz isso por você são funções de uma biblioteca padrão.

Suponha que você queira ler um único caractere, mas não quer usar o scanf(). Isso pode ser feito usando a função getchar(). A função putchar() aceita um argumento de entrada, cujo valor será impresso como caracter:

```
#include <stdio.h>

main()
{
   char ch;

   printf("Digite algum caracter: ");

   ch = getchar();

   printf("\n A tecla pressionada eh %c.\n", ch);

}
```

O Resultado deste programa na tela é:

```
Digite algum caracter: A
 A tecla pressionada eh A.
```

Outro exemplo:

```
#include <stdio.h>

main()
{
   char ch;

   printf("Digite outro caracter: ");

   ch = getchar();

   putchar(ch);

}
```

O Resultado deste programa na tela é:

```
Digite outro caracter: B
B
```

6 Ordem sequencial de execução de sentenças o comando condicional: `if` and `if - else`

A execução de um programa C começa com a função `main()`. Em todos os exemplos que vimos até este momento, sentenças são executadas sequencialmente. A ordem sequencial de execução de senteças pode ser alterada se certas condições forem satisfeitas durante a execução do programa. Isto é chamado *desvio condicional*.

Todas as linguagens de programação oferecem comandos para o desvio condicional. O mais simples é a senteça `if`. Em C, ele tem o formato:

`if` (*expressao*)
sentenca

Quando uma sentença `if` é encontrada em um programa,

1. O teste na *expressao* em parênteses é avaliada.
2. Se o valor da expressão de teste for DIFERENTE de zero, a *sentença* que segue a expressão de teste é executada.

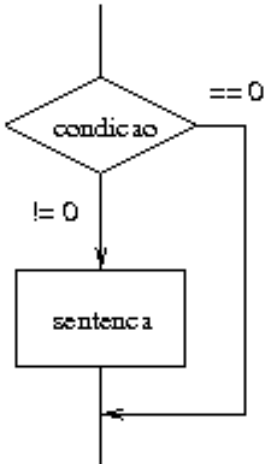


Figura 1: O comando `if`

Considere o seguinte exemplo que converte uma fração digitada pelo usuário (numerador e denominador) em decimal e imprime o resultado:

```
#include <stdio.h>

main(void)
{
  int a, b;

  printf("Entre com uma  fracao (numerador and denominador): ");
  scanf("%d %d", &a, &b);

  printf("A fracao em decimal eh %f\n", 1.0 * a / b);
}
```

No exemplo acima, escrevemos `1.0 * a / b`, já que `a` e `b` são do tipo `int`, e portanto `a / b` é uma divisão de inteiros e a parte fracional do resultado seria truncado, o que certamente não é o que desejamos.

Voce vê algo errado neste programa ? Uma coisa a ser notada é que se o usuário digitar um denominador igual a `0`, nós teremos um erro de execução, já que o programa tentaria executar uma divisão por zero. O que é necessário fazer é testar se o denominador é igual a zero e dividir só no caso dele for diferente de zero. Poderíamos reescrever o programa acima da seguinte forma:

Exemplo 1:

```
#include <stdio.h>

main(void)
{
  int a, b;

  printf("Entre com uma  fracao (numerador e denominador): ");
  scanf("%d %d", &a, &b);

  if (b != 0)
    printf("A fracao em decimal eh %f\n", 1.0 * a / b);
}
```

Exemplo 2: Programa que lê dois números e ordena o par caso o primeiro número digitado for maior que o segundo.

```
#include <stdio.h>

main() {
  int num1, num2, aux;

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

  if (num1 > num2) {
    aux = num1;
    num1 = num2;
    num2 = aux;
    printf("Trocou \n");
  }

  printf("Os numeros ordenados: %d %d\n", num1, num2);

}
```

O programa do Exemplo 1 acima ficaria ainda melhor se ao invés de não fazer nada no caso do denominador ser zero, imprimirmos uma mensagem de erro ao usuário, explicando o que há de errado.

A sentença em C que permite fazermos isso é o `if - else`. O formato do `if-else` é:

```
if (expressao)
        sentenca1
else
        sentenca2
```

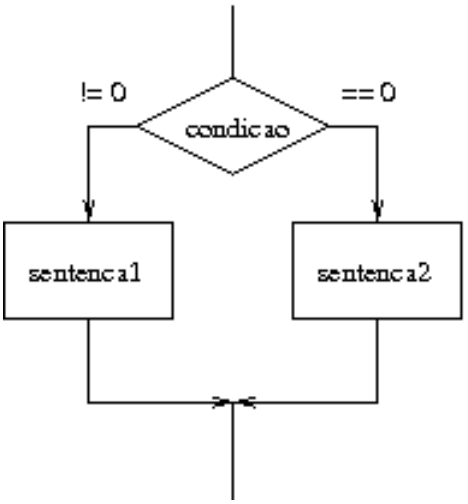


Figura 2: O comando `if-else`

Primeiro, a $expressao$ (que usualmente chamamos de condição) é avaliada. Caso a condição seja verdadeira (o que é equivalente a dizer que o valor é diferente de zero), entao a $sentenca_1$ é executada. Caso contrário, a $sentenca_2$ é executada.

Note que uma *sentença* pode ser simples ou composta. Se você quiser agrupar diversas sentenças para serem executadas, você pode colocá-las entre chaves (`{` e `}`).

Por hora, vamos continuar com nosso exemplo simples e torná-lo mais explicativo:

Exemplo 3:

```
#include <stdio.h>

main(void)
{
  int a, b;

  printf("Entre com uma fracao (numerador and denominador): ");
  scanf("%d %d", &a, &b);

  if (b != 0)
    printf("A fracao decimal e  %f\n", 1.0 * a / b);
  else
    printf("Erro: denominador zero!\n");
}
```

Exemplo 4: Considere agora o exemplo já visto que pede que um usuário entre com um número e verifique se o número é par. Porém agora, queremos que o programa imprima "o numero e par" ou "o numero e impar".

```
#include <stdio.h>

main(void)
{
  int num;
```

```
  /* obtem um numero do usuario */
  printf("Entre com um inteiro: ");
  scanf("%d", &num);

  /* imprime uma mensagem dizendo se o numero e par ou impar */
  if (num % 2 == 0)
    printf("O numero e par.\n");
  else
    printf("O numero e impar.\n");

}
```

6.1 Um erro comum

É muito frequente utilizar o operador relacional == em expressões condicionais da sentença `if`. Por exemplo:

```
    int  saldo = 2000;

    if (saldo == 1)
       printf("Voce esta quebrado! \n");
    else
       printf("Seu saldo e %d\n", saldo);
```

Como a sentença `saldo = 2000` inicializa o valor da variável `saldo` com `2000`, a expressão `saldo == 1` tem valor `0`. Portanto, a senteça que segue o `else` será executada, e a mensagem

```
Seu saldo e 2000
```

será impressa.

Agora, suponha que, devido a um erro, você tenha colocado = ao invés de ==:

```
  int saldo = 2000;

  if (saldo = 1)
     printf("Voce esta quebrado! \n");
  else
     printf("Seu saldo e %d\n", saldo);
```

Agora, a expressão `saldo = 1` tem valor `1`. Portanto, a sentença que segue o `if` será executada, e a mensagem

```
Voce esta quebrado!
```

será impressa. Além disso, a atribuição causará um efeito colateral, e alterará o valor de `saldo` para `1`.

Tal uso do operador de atribuição não é ilegal, e não será detectado pelo compilador como erro. Portanto, *tome cuidado com o uso de atribuição no lugar de igualdade*. Tal erro é muito comum, e não é fácil de achar.

Como regra geral, NÃO utilize atribuições dentro de outras sentenças.

7 Aninhando senteças `if` e `if-else`

Como era de se esperar, é possível colocar uma sentença condicional dentro de outra. Por exemplo, se quisermos imprimir uma mensagem apropriada caso um número seja positivo ou negativo e par ou ímpar, nós poderíamos escrever o seguinte:

```
#include <stdio.h>

main(void)
{
  int num;

  /* Obtem um numero do usuario */
  printf("Entre com um inteiro: ");
  scanf("%d", &num);

  /* Imprime uma mensagem dizendo se o numero e positivo ou
     negativo, positivo ou negativo. */
  if (num >= 0) {
    if (num % 2 == 0)
      printf("O numero e par e positivo\n");
    else
      printf("O numero e impar e positivo\n");
  }
  else {
    if (num % 2 == 0)
      printf("O numero e par e negativo\n");
    else
      printf("O numero e impar e negativo\n");
  }
}
```

7.1 A ambigüidade do `else`

O aninhamento de sentenças `if-else` sem usar chaves (`{` e `}`) para delimitar o bloco de senteças a ser executado pode trazer efeitos indesejados.

Há uma regra simples para determinar qual `if` está associado a qual `else`.

Regra de associação: Um `else` está associado com a última ocorrência do `if` sem `else`.

O exemplo seguinte está errado porque associa o `else` ao `if` "incorreto":

```
#include <stdio.h>

main(void)
{
  int num;

  /* Obtem um numero do usuario */
  printf("Entre com o numero de peras: ");
  scanf("%d", &num);
```

```
  /* Imprime uma mensagem dizendo se o numero de peras e 0 ou 1
     (*** isto esta' errado !!  ***) */
  if (num != 0)
    if (num == 1)
      printf("Voce tem uma pera.\n");
  else
    printf("Voce nao tem nenhuma pera.\n");
}
```

Neste exemplo, o `if` tem o seguinte significado, segundo a regra de associação:

```
#include <stdio.h>

main(void)
{
  int num;

  /* Obtem um numero do usuario */
  printf("Entre com o numero de peras: ");
  scanf("%d", &num);

  /* Como a sentenca if e' vista pelo compilador */
  if (num != 0)
    if (num == 1)
      printf("Voce tem uma pera.\n");
    else
      printf("Voce nao tem nenhuma pera.\n");
}
```

Para evitar este problema, chaves (`{` e `}`) devem ser usadas para tirar a ambiguidade. O exemplo abaixo mostra como as chaves podem ser inseridas para corrigir o programa acima.

```
#include <stdio.h>

main(void)
{
  int num;

  /* Obtem um numero do usuario */
  printf("Entre com o numero de peras: ");
  scanf("%d", &num);

  /* Como corrigir o problema (este programa funciona) */
  if (num != 0) {
    if (num == 1)
      printf("Voce tem uma pera.\n");
  }  else
    printf("Voce nao tem nenhuma pera.\n");
}
```

Exercício 1: Faça um programa que leia 3 números e imprima o maior.

```
#include <stdio.h>

main(void)
{
  int a, b, c, maior;

  printf("Entre com os tres numeros: ");
  scanf("%d %d %d", &a, &b, &c);

  if (a > b)
    maior = a;
  else
    maior = b;

  if (maior < c)
    maior = c;

  printf("O Maior numero eh %d\n", maior);

}
```

8 Operadores Lógicos

Todos os programas até agora consideraram `if` com condições de teste simples. Alguns exemplos de testes simples: `b != 0`, `contador <= 5`. Estas expressões testam uma condição. Portanto, quando mais de uma condição precisa ser testada, precisamos usar sentenças `if` e `if-else` aninhadas.

A lingugem C, assim como a maioria das linguagens de programação de alto nível suportam *operadores lógicos* que podem ser usados para criar *operações lógicas* mais complexas, combinando condições simples. O valor de uma expressão lógica é ou VERDADEIRO ou FALSO. Lembre que não há constantes lógicas VERDADEIRO e FALSO em C; em expressões lógicas `0` é interpretado como FALSO, e qualquer valor diferente de zero é interpretado como VERDADEIRO.

Os operadores lógicos são

`!`	NÃO lógico, operação de negação (operador unário)
`&&`	E lógico, conjunção (operador binário)
`||`	OU lógico, disjunção (operador binário).

Por exemplo, se quisermos testar se um número `num` é positivo e par, e imprimir uma mensagem como no exemplo anterior, podemos escrever:

```
if (num >= 0)
   if (num % 2 == 0)
      printf("Numero par nao negativo.\n");
```

Com os operadores lógicos isso pode ser simplificado:

```
if ((num>=0) && (num%2 == 0))
    printf("Numero par nao negativo.\n");
```

A operação de negação, `!`, pode ser usado da seguinte forma:

`!` *expressão lógica*: O valor é a negação lógica da expressão dada. Por exemplo:

`!0` é `1`
`!1` é `0`

Nós podemos usar o operador de negação lógica e escrever o exemplo acima como:

```
if (num>0 && !(num%2))
      printf("Numero par nao negativo.\n");
```

Os dois operadores binários operam sobre duas expressões lógicas e tem o valor `1` (verdadeiro) or `0` (falso). Os exemplos abaixo mostram o seu uso:

`a==0 && b==0` (verdadeiro se ambos `a == 0` e `b == 0`, portanto se `a` e `b` são `0`)
`a==0 || b==0` (verdadeiro se pelo menos uma das variáveis `a` or `b` for `0`)

Uma expressão usando `&&` é verdadeira somente se ambos os operadores forem verdadeiros (não zero).

Uma expressão usando `||` é falsa somente se ambos os operadores forem falsos (zero).

Verifique na Tabela 3 o resultado do uso de operadores lógicos:

$expr_1$	$expr_2$	$expr_1$ `&&` $expr_2$	$expr_1$ `||` $expr_2$
$verdadeiro$	$verdadeiro$	$verdadeiro$	$verdadeiro$
$verdadeiro$	$falso$	$falso$	$verdadeiro$
$falso$	$verdadeiro$	$falso$	$verdadeiro$
$falso$	$falso$	$falso$	$falso$

Tabela 3: Resultado de uso de Operadores Lógicos

A precedência do operador de negação lógica é a mais alta (no mesmo nível que o "-" unário). A precedência dos operadores lógicos binários é menor que a dos operadores relacionais, e mais alta que a operação de atribuição. O `&&` tem precedência mais alta que o `||`, e ambos associam da esquerda para a direita (como os operadores aritméticos).

Como a precedência dos operadores lógicos é menor que a dos operadores relacionais, não é necessário usar parênteses em expressões como:

```
x >= 3 && x <= 50
x == 1 || x == 2 || x == 3
```

A Tabela 4 mostra o quadro completo de precedência de operadores aritméticos, relacionais e lógicos.

No próximo exemplo, o programa verifica se as três variáveis `lado1`, `lado2`, e `lado3`, podem ser lados de um triângulo reto. Nós usamos o fato que os três valores devem ser positivos, e que o quadrado de um dos lados deve ser igual a soma dos quadrados dos outros lados (Teorema de Pitágoras) para determinar se o triângulo é reto.

Operador	Associatividade
`()`	esquerda para direita
`!  -  ++  --  (cast)  &` (unários)	direita para esquerda
`*  /  %`	esquerda para direita
`+  -`	esquerda para direita
`<  <=  >  >=`	esquerda para direita
`==  !=`	esquerda para direita
`&&`	esquerda para direita
`||`	esquerda para direita
`=  +=  -=  *=  /=  %=`	direita para esquerda

Tabela 4: Precedência e associatividade de operadores

```
#include <stdio.h>

main() {

  int lado1, lado2, lado3;
  int s1, s2, s3;

  printf(``Entre com o tamanho dos lados do triangulo: ``);
  scanf(``%d %d %d'', &lado1, &lado2, &lado3);

  /* calcula o quadrado dos lados */
  s1 = lado1*lado1;
  s2 = lado2*lado2;
  s3 = lado3*lado3;

  /* testa a condicao para um triangulo reto */

  if ( lado1>0 && lado2>0 && lado3 > 0 ) {
    if (s1==s2+s3 || s2==s1+s2 || s2==s1+s3) ) {
    printf("Triangulo reto!\n");
  }
  else {
    printf("Nao pode ser um triangulo!\n");
  }
}
```

Na utilização de expressões lógicas, as seguintes identidades são úteis. Elas são chamadas de *Lei de DeMorgan*:

`!(x && y)` é equivalente a `!x || !y`

e

`!(x || y)` é equivalente a `!x && !y`

9 Exemplos

9.1 IF - ELSE

Assuma as seguintes declaraçõoes de variáveis:

```
int x = 4;
int y = 8;
```

O que é impresso pelos seguintes programas ?

1.
```
if (y = 8)
    if (x = 5)
        printf( "a\n" );
    else
        printf( "b\n" );
printf( "c\n" );
printf( "d\n" );

==> a c d
```

2. mude = para ==

```
==> b c d
```

3. altere o programa acima para produzir a seguinte saida:

   - Assuma `x = 5` e `y = 8`
     (a) `a`
     (b) `a d`
   - Assuma `x = 5` e `y = 7`
     (a) `b c d`

9.2 Operadores lógicos

O que é impresso pelas seguintes sentenças:

1. Assuma `x = 5` e `y = 8`.

```
if (x == 5 && y == 8)
    printf( "a\n" );
else
    printf( "b\n" );      ==> a
```

2. Assuma `x = 4` e `y = 8`.

```
if (x == 5 || y == 8)
    printf( "a\n" );
else
    printf( "b\n" );      ==> a


if !(x == 5 || y == 8)
    printf( "a\n" );
else
    printf( "b\n" );      ==> b  equiv. \mbox{(x != 5 && y != 8)}
```

```
if !(x == 5 && y == 8)
    printf( "a\n" );
else
    printf( "b\n" );      ==> a  equiv. \mbox{(x != 5 || y != 8)}
```

3. Precedência: `!  > && > ||`

```
if (x == 5 ||  y == 8 && z == 10)

equiv.

if (x == 5 || (y == 8 && z == 10))
```

10 A construção `else-if`

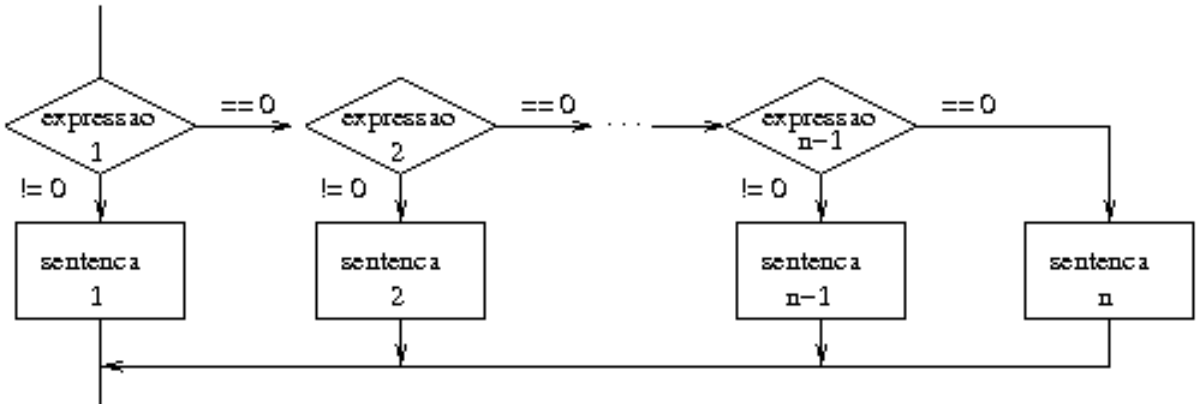
Embora ela não seja um tipo diferente de sentença, a seguinte construção é bastante comum para programar decisões entre diversas alternativas:

```
if (expressao1)
        sentenca1
else if (expressao2)
        sentenca2
else if (expressao3)
        sentenca3
⋮
else if (expressao_{n-1})
        sentenca_{n-1}
else
        sentenca_n
```

As expressões lógicas são avaliadas em ordem, começando com a $expressao_1$. Se uma das expressões for verdadeira, a sentença associada será executada. Se nenhuma for verdadeira, então a sentença, $sentenca_n$, do último `else` será executada como opção *default*. Se a opção *default* não for necessária, então a parte

```
else
        sentenca_n
```

pode ser removida.



Exemplo 9: O seguinte exemplo mostra um `else-if` de três opções. O programa lê dois números e diz se eles são iguais ou se o primeiro número é menor ou maior que o segundo.

```
#include <stdio.h>

main(void)
{
  int num1, num2;

  /* obtem 2 numeros do usuario */
  printf("Entre um numero: ");
  scanf("%d", &num1);
  printf("Entre com um outro numero: ");
  scanf("%d", &num2);

  /* mostra a mensagem de comparacao */
  if (num1 == num2)
    printf("Os numeros sao iguais\n");
  else if (num1 < num2)
    printf("O primeiro numero e menor\n");
  else
    printf("O primeiro numero e maior\n");
}
```

No programa acima, se `(num1 == num2)` for verdadeiro, então os números são iguais. Senão, é verificado se `(num1 < num2)`. Se esta condição for verdadeira, então o primeiro número é menor. Se isso não for verdadeiro, então a única opção restante é que o primeiro número é maior.

Exemplo 10: Este programa lê um número, um operador e um segundo número e realiza a operação correspondente entre os operandos dados.

```
#include <stdio.h>

main(void)
{
  float num1, num2;
  char op;

  /* obtem uma expressao do usuario */
```

```
  printf("Entre com numero operador numero\n");
  scanf("%f %c %f", &num1, &op, &num2);

  /* mostra o resultado da operacao */
  if (op == '+')
    printf(" = %.2f", num1 + num2);
  else if (op == '-')
    printf(" = %.2f", num1 - num2);
  else if (op == '/')
    printf(" = %.2f", num1 / num2);
  else if (op == '*')
    printf(" = %.2f", num1 * num2);
  else
    printf(" Operador invalido.");
  printf("\n");
}
```

Exemplos da execução deste programa:

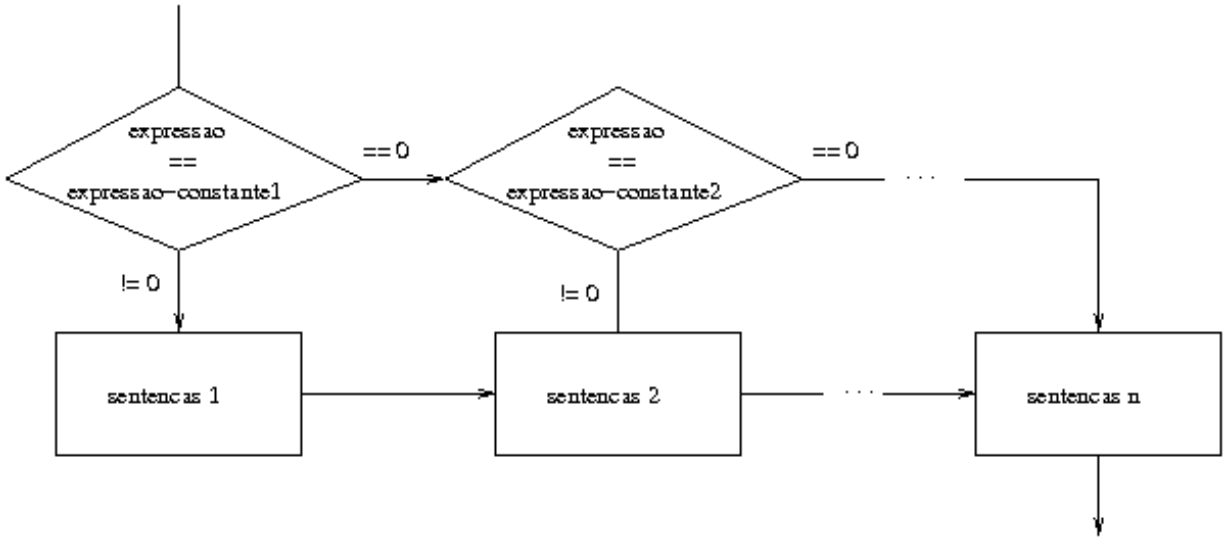
```
Entre com numero operador numero:
5 * 3.5
 = 17.50

Entre com numero operador numero:
10 + 0
 = 10.00

Entre com numero operador numero:
10 x 5.0
 Operador invalido.
```

11 A sentença `switch`

A sentença `switch` é outra maneira de fazer decisões múltiplas. Ele pode ser usado para testar se uma dada expressão é igual a um valor constante e, dependendo do valor, tomar determinadas ações.

O formato da sentença `switch` é:

```
switch (expressao) {
        case expressao-constante 1:
                sentencas 1
        case expressao-constante 2:
                sentencas 2
                ⋮
        default:
                sentencas n
}
```



A sentença `switch` primeiro avalia a *expressão*. Se o valor da expressão for igual a uma das *expressões constantes*, as *sentenças* que seguem o `case` são executados. Se o valor da *expressão* não for igual a nenhuma das constantes, as sentenças que seguem `default` são executadas.

As *sentenças* que seguem o `case` são simplesmente uma lista de sentenças. Esta lista pode conter mais de uma sentença e não é necessário colocá-las entre chaves (`{` e `}`). A lista de sentenças também pode ser vazia, isto é, você pode não colocar nenhuma sentença seguindo o `case`.

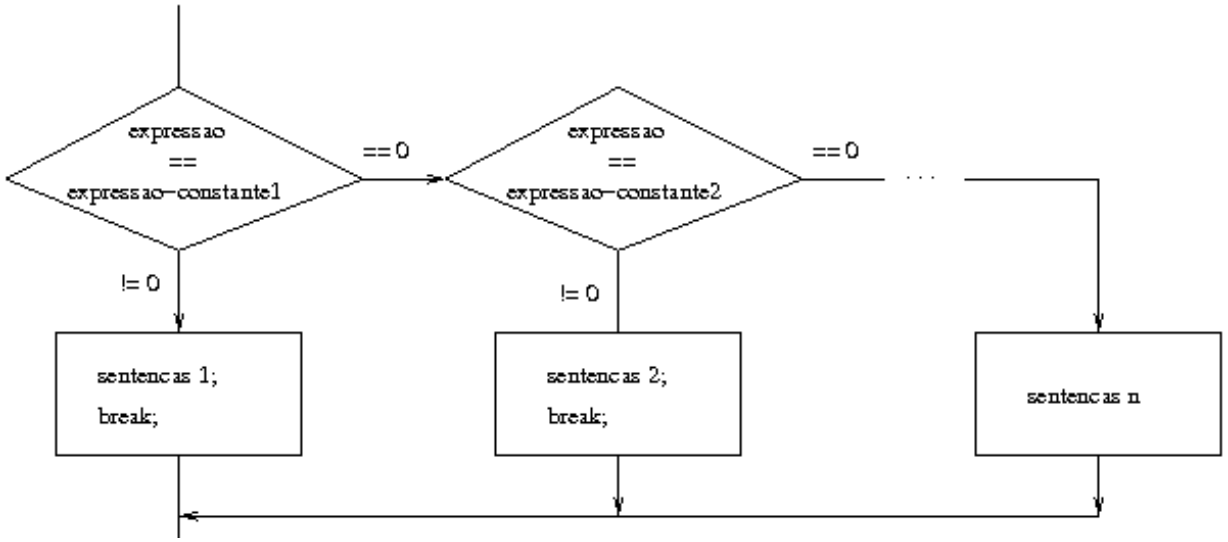
Também não é obrigatório colocar o `default`. Só o use quando for necessário.

Note no diagrama acima que TODAS as sentenças que seguem a constante com o valor igual ao da expressão serão executados. Para que se execute APENAS as sentenças que seguem o `case` que seja igual ao valor da expressão precisamos usar a sentença `break`, que veremos em seguida.

12 A sentença `break`

O `break` faz com que todas as sentenças que o seguem dentro da mesma sentença `switch` sejam ignorados. Ou seja, colocando a sentença `break` no final de uma sentença `case` faz com que as sentenças que seguem os `case`s subsequentes não sejam executadas. Em geral, é este o comportamento desejado quando se usa o `switch`, e `case`s sem o `break` no final são de pouca utilidade. Portanto, o uso de sentenças `case` sem o `break` devem ser evitados e quando utilizados devem ser comentados ao lado com algo como `/* continua proxima sentenca - sem break */`.

Com a sentença `break` o diagrama de fluxo fica:



Note a similaridade com o diagrama da sentença `else-if` e a diferença com o diagrama da sentença `switch` acima.

O próximo programa tem a mesma função de calculadora do programa anterior, porém utilizando a sentença `switch`.

Exemplo 11:

```
#include <stdio.h>

main(void)
{
  float num1, num2;
  char op;

  printf("Entre com numero operador numero:\n");
  scanf("%f %c %f", &num1, &op, &num2);

  switch (op) {
  case '+':
    printf(" = %.2f", num1 + num2);
    break;
  case '-':
    printf(" = %.2f", num1 - num2);
    break;
  case '*':
    printf(" = %.2f", num1 * num2);
    break;
  case '/':
    printf(" = %.2f", num1 / num2);
    break;
  default:
    printf(" Operador invalido.");
    break;
  }
  printf("\n");
```

```
}
```

Como mencionado anteriormente, é possível não colocar nenhuma sentença seguindo um `case`. Isso é útil quando diversas sentenças `case` (diversas constantes) têm a mesma ação.

Por exemplo, podemos modificar o programa acima para aceitar `x` e `X` para multiplicação e `\` para divisão. O programa fica então:

```
#include <stdio.h>

main(void)
{
  float num1, num2;
  char op;

  printf("Entre com numero operador numero:\n");
  scanf("%f %c %f", &num1, &op, &num2);

  switch (op) {
  case '+':
    printf(" = %.2f", num1 + num2);
    break;
  case '-':
    printf(" = %.2f", num1 - num2);
    break;
  case '*':
  case 'x':
  case 'X':
    printf(" = %.2f", num1 * num2);
    break;
  case '/':
  case '\\':
    printf(" = %.2f", num1 / num2);
    break;
  default:
    printf(" Operador invalido.");
    break;
  }
  printf("\n");
}
```

Exercício 2: Ler mes e ano e imprimir o numero de dias do mes no ano digitado.

```
#include <stdio.h>

main() {
  int mes, ano, numDias;

  printf("Entre com mes e ano (mm aa):");
  scanf( "%d %d", &mes, &ano);
  if( mes < 1 || mes > 12 || ano < 0 || ano > 99 )
    printf("mes ou ano invalido\n");
```

```
  else {
    switch( mes ){
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
      numDias = 31;
      break;
    case 2:
      if( ano % 4 == 0 )
        numDias = 29;
      else
        numDias = 28;
      break;
    default:
      numDias = 30;
    }
    printf("%2d/%2d tem %d dias\n", mes, ano, numDias);
  }
}
```

13 Estruturas de Repetição

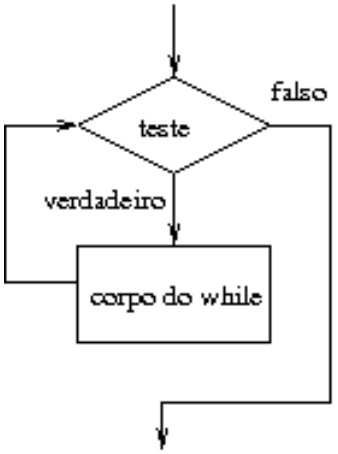
A linguagem C possui comandos para repetir uma sequência de instruções. Estas **estruturas de repetição**, também conhecidas como **laços** (do inglês ***loops***). A primeira construção que veremos é o while, seguida de for e de do ... while.

13.1 O comando de repetição while

O comando de repetição while tem duas partes: a *expressão de teste* e o *corpo da repetição*. O formato do while é:

while (*expressão teste*)
corpo da repetição

A *expressão teste* é inicialmente avaliada para verificar se o laço deve terminar. Caso a expressão seja verdadeira (isto é, diferente de 0 (zero)), o *corpo da repetição* é executado. Depois desta execução, o processo é repetido a partir da *expressão teste*. O *corpo do laço*, por sua vez, pode ser uma sentença simples ou composta.



O exemplo abaixo mostra o uso do comando de repetição while:

```
int contador = 0;

while( contador < 5 )
  printf( "contador = %d\n", contador++ );

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

Saída:

```
contador = 0
contador = 1
contador = 2
contador = 3
contador = 4
ACABOU !!!!
```

Neste exemplo, a expressão de teste é contador < 5, e o corpo do laço é a sentença printf().

Se examinarmos cuidadosamente este exemplo, veremos que a variável contador é inicializada com 0 (zero) quando é definida. Depois disso, a expressão de teste é verificada e, como 0 < 5 é verdadeiro, o corpo da repetição é executado. Assim, o programa imprime contador = 0, e incrementa contador de

um (através do pós-decremento indicado no argumento de printf()). Em seguida, a expressão de teste é verificada novamente e todo o processo se repete até que contador seja 4 e contador = 4 seja impresso.

Depois disso, contador é incrementado para 5 e o teste é executado. Mas desta vez, 5 < 5 é falso, então o laço não continua. A execução do programa continua na sentença que segue o laço (no caso, imprimir a frase ACABOU !!!).

Após a execução do while, a variável contador tem valor 5.

No exemplo acima, há uma sentença simples no corpo da repetição. Quando este for definido por uma sentença composta (bloco), não se deve esquecer de usar as chaves ({ e }) para delimitar o bloco da sentença composta.

O exemplo seguinte mostra um uso mais apropriado do comando while: Em situações onde o número de repetições não é conhecido antes do inicío do comando while:

Exemplo 1: Este programa pede números ao usuário até que a soma de todos os números digitados for pelo menos 20.

```
#include <stdio.h>

main( ){

  int total = 0, num;

  while( total < 20 ) {
    printf( "Total = %d\n", total );

    printf( "Entre com um numero: " );
    scanf( "%d", &num );

    total += num;
  }

  printf( "Final total = %d\n", total );
}
```

Exemplo de saída:

```
Total = 0
Entre com um numero: 3
Total = 3
Entre com um numero: 8
Total = 11
Entre com um numero: 15
Final total = 26
```

Inicialmente, é dado o valor 0 à variável total, e o teste é verdadeiro (0 < 20). Em cada iteração, o total é impresso e o usuário digita um número que é somado a total. Quanto total for maior ou igual a 20, o teste do while torna-se falso, e a repetição termina.

13.2 A Estrutura de Repetição do...while

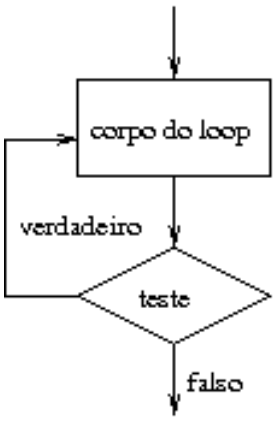
Há outro comando de repetição em linguagem C. O do...while é bastante parecido com while, com a diferença que a expressão de teste é avaliada DEPOIS que o corpo da repetição é executado.

O formato do do...while é:

do

corpo da repetição

while (*expressão teste*)



O exemplo abaixo usa o do...while:

```
int contador = 0;

do {
   printf( "contador = %d\n", contador++ );
} while( contador < 5 )

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

A execução deste programa é idêntico ao primeiro exemplo mostrado para o comando while, com a expressão de teste mudada para o final.

Saída:

```
contador = 0
contador = 1
contador = 2
contador = 3
contador = 4
```

O do...while é usado quando o corpo da repetição deve ser executado pelo menos uma vez. Um exemplo comum disto é o processamento da entrada de um programa.

Exemplo 2: Neste exemplo, o teste do laço é baseado no valor digitado pelo usuário. O laço deve ser executado pelo uma vez antes que o teste sobre o valor seja executado.

```
#include <stdio>

main( ){

  int num;

  printf( "Entre com um numero par:\n" );

  do{
```

```
    scanf( "%d", &num );
  } while( num % 2 != 0 );

  printf( "Obrigado.\n" );
}
```

Exemplo de execução:

```
Entre com um numero par:
3
1
5
4
Obrigado.
```

Neste caso, o valor da variável num é digitado pelo usuário. Depois disso, o teste é executado para verificar se o número é par (o teste `num % 2 != 0` é falso se num é par já que o resto da divisão de qualquer número par por 2 é zero).

É possível escrever o programa acima usando while:

```
#include <stdio>

main( ){

  int num = 1;       /* Atribui um numero impar a  num */

  printf( "Entre com um numero par:\n" );

  while( num % 2 != 0 ){
    scanf( "%d", &num );
  }
  printf( "Obrigado.\n" );
}
```

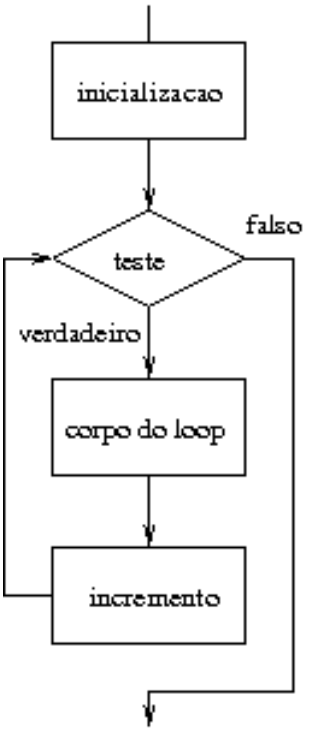
O problema com este programa é que a variável num deve ser inicializada com um valor que torne o teste do laço verdadeiro. Neste exemplo, é simples encontrar tal valor. Para uma expressão teste mais complicada, isso pode não ser tão fácil.

13.3 A Extrutura de Repetição for

Há 4 partes no laço for: *inicialização*, *expressão de teste*, *expressão de incremento* e o *corpo do laço*. O formato do laço for é:

for (*inicialização*; *expressão de teste*; *incremento*){
corpo da repetição
}

A *inicialização* é executada uma única vez no início do laço. A *expressão teste* é então avaliada para verificar se o laço deve terminar. Caso a expressão seja verdadeira (isto é, diferente de Zero), o *corpo da repetição* é executado. Depois desta execução, a expressão de *incremento* é executada e o processo é repetido a partir da *expressão teste*. O *corpo da repetição*, por sua vez, pode ser uma sentença simples ou composta.



Veja abaixo um exemplo simples do laço for:

```
int contador;

for( contador = 0; contador < 5; contador++ )
    printf( "contador = %d\n", contador );

printf{"ACABOU !!!!\n");
```

Saída do programa:

```
contador = 0
contador = 1
contador = 2
contador = 3
contador = 4
ACABOU !!!!
```

Se você examinar cuidadosamente este exemplo, poderá ver precisamente o que está acontecendo. Primeiro, a inicialização é executada, que é a sentença contador = 0. Isso modifica o valor da variável contador para 0. Então, o teste é executado. como 0 < 5 é verdadeiro, o laço continua. Assim, o corpo da repetição é executado, imprimindo a primeira linha da saída, contador = 0. Depois disso, o incremento é executado, que é a sentença contador++, que altera o valor da variável contador para 1.

Esta é a 1ª iteração do laço. Então, o teste é executado novamente (como 1 < 5 é verdadeiro, o laço continua), o corpo da repetição mostra contador = 1, e contador é incrementado novamente.

Este processo continua até que contador seja 4 e contador = 4 seja impresso. Depois disso, contador é incrementado para 5 e o teste é executado. Mas desta vez, 5 < 5 é falso, então o laço não continua. A execução do programa continua na sentença que segue o laço (no caso, imprimir a frase ACABOU !!!).

Após a execução do laço, a variável contador tem valor 5.

Ao invés de usar o teste contador < 5, você poderia também ter usado a expressão contador <= 4. O resultado seria o mesmo. Use a expressão que você preferir. Outra expressão que também poderia ter sido usada é contador != 5. Porém esta expressão torna o programa menos legível (não é tão evidente que o valor

de contador está sendo incrementado até atingir o valor 5). Além disso, isso poderia causar problemas se mudássemos a inicialização para um valor maior que 5. Por exemplo, se a inicialização for contador = 25 e a expressão teste for contador != 5 o laço nunca terminaria, pois o contador começa com 25 e a cada iteração o valor é incrementado, o que nunca tornaria o teste falso.

Também poderíamos ao invés de usar contador++ como a expressão de incremento, usar ++contador, contador += 1 e contador = contador + 1. O resultado seria o mesmo (neste caso, o uso de pós- e pré-incremento não faz diferença).

Se você quisesse incrementos de dois, você poderia escrever contador += 2 (ou contador = contador + 2).

13.3.1 Diversas sentenças dentro de um laço

Como no comando while, o corpo da repetição pode ser definido por uma sentença simples ou composta. No caso de uma sentença composta (bloco), não se deve esquecer de usar as chaves ({ e }) para delimitar o bloco da sentença composta.

Em um for também podemos ter mais de uma expressão de inicialização ou incremento. Nestes caso, elas devem ser separadas por vírgula (,) o que é ilustrado no exemplo abaixo:

Exemplo 3:

```
#include <stdio.h>

#include <stdio.h>

main( void ){

  int contador, total;

  for( contador = 0, total = 0; contador < 10; contador++ ){
    total += contador;
    printf( "contador = %d, total = %d\n", contador, total );
  }
}
```

Saída:

```
contador = 0, total = 0
contador = 1, total = 1
contador = 2, total = 3
contador = 3, total = 6
contador = 4, total = 10
contador = 5, total = 15
contador = 6, total = 21
contador = 7, total = 28
contador = 8, total = 36
contador = 9, total = 45
```

No exemplo acima, contador = 0, total = 0 é a inicialização, contador < 10 é a expressão teste, e contador++ é a expressão de incremento.

Exemplo 4: Um programa que imprime todos os números entre 30 e 5 (nesta ordem) divisíveis por 3, e no final imprime sua soma.

```
#include <stdio.h>

main( ){

  int i, soma = 0;

  for( i = 30; i >= 5; i-- ){
    if( (i % 3) == 0 ){
      printf( "\t%2d\n", i );
      soma += i;
    }
  }
  printf( "\t soma = %d\n", soma );
}
```

Saída do programa:

```
30
27
24
21
18
15
12
 9
 6
soma = 162
```

13.3.2 Laços aninhados

É possível colocar um laço dentro de outro (laço aninhado).

Exemplo 5:

```
#include <stdio.h>

main( ){

  int linha, coluna;

  for( linha = 1; linha < 5; linha++ ){
    for( coluna = 1; coluna < 5; coluna++ )
      printf( "%3d", linha * coluna );

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

Saída:

```
  1  2  3  4
  2  4  6  8
  3  6  9 12
  4  8 12 16
```

No exemplo acima, para cada iteração do laço externo, o laço interno imprime uma linha com números e depois pula de linha.

Exemplo 6: Este exemplo é parecido com o anterior, exceto que o printf() é colocado dentro do laço interno. Como era de se esperar uma nova linha é impressa após cada valor ao invés de ser depois de 4 valores.

```
#include <stdio.h>

main( ){

  int linha, coluna;

  for( linha = 1; linha < 5; linha++ ){
    for( coluna = 1; coluna < 5; coluna++ ){
      printf( "%3d", linha * coluna );
      printf( "\n" );
    }
  }
}
```

Saída:

```
  1
  2
  3
  4
  2
  4
  6
  8
  3
  6
  9
 12
  4
  8
 12
 16
```

Exemplo 7:

```
#include <stdio.h>

main( ){

  int linha, coluna;

  printf("\n");
  for( linha = 1; linha < 10; linha++ ){
    printf( "\t" );
```

```
    for( coluna = 1; coluna < linha; coluna++ )
      printf( "*" );
    printf( "\n" );
  }
}
```

Saída:

```
*
**
***
****
*****
******
*******
********
```

13.4 Estilo de formatação para estruturas de repetição

A regra principal é ser **consistente**. Assim, seu programa será mais legível.

13.4.1 Colocação das chaves

Há três estilos comuns de colocar as chaves:

```
while (expressao)
   {
   sentenca;
   }

while (expressao)
{
   sentenca;
}

while (expressao) {
   sentenca;
}
```

APENAS UM DESTES ESTILOS deve ser consistentemente usado para as sentenças for, while e do ... while. Use o estilo com o qual você se sentir mais confortável.

13.4.2 Necessidade ou não das chaves

Foi mencionado anteriormente que as chaves ({ e }) podem ser omitidas quando o corpo da repetição contiver apenar uma sentença. Por exemplo:

```
while( i < 5 )
   i++;
```

Embora as chaves possam ser omitidas, há uma única razão para colocá-las sempre:

```
while( i < 5 ) {
   i++;
}
```

Quando você adicionar algo ao programa, você poderá adicionar uma sentença para um laço com apenas uma sentença. Se você fizer isso, é vital que você também adicione chaves. Se você não fizer isso, a segunda sentença do laço não será considerada como parte do laço. Por exemplo:

```
while( i < 5 )
   i++;
   j++;
```

é na verdade o mesmo que:

```
while( i < 5 )
   i++;
j++;
```

enquanto a intenção era na realidade:

```
while( i < 5 ) {
   i++;
   j++;
}
```

13.4.3 Uso de espaço em branco

A outra questão de formato é se deve ser colocado um espaço em branco depois do for e while e antes do abre parênteses ((). Por exemplo:

```
for (i=0; i<5; i++)
```

ou

```
for(i=0; i<5; i++)
```

ou

```
for( i=0; i<5; i++ )
```

Isto também é uma escolha pessoal. Porém seja consistente em sua escolha !

13.5 Lendo o teclado

Como vimos nas Notas de Aula # 06[2], há situações em que a função scanf() não se adapta às necessidades do programa, pois é necessário pressionar **RETURN** para que scanf() proceda com a leitura. Vimos naquela ocasião uma função que lê um caracter no instante em que ele é digitado seguido de um **RETURN** : A função getchar(). Veremos aqui exemplos mais práticos de seu uso, bem como de duas novas funções: getche() e getch().

[2] ../teoria_06/index.html#SECTION00030000000000000000

13.5.1 Lendo o teclado usando getchar()

getchar() é uma função da biblioteca padrão stdio. Cada vez que é chamada, a função lê um caractere teclado; getchar começa a ler depois que a tecla **RETURN** é digitada no final de uma sequência de caracteres (dizemos que a entrada para a função getchar() está no *buffer*). A função getchar()retorna um valor, o caractere lido (mais precisamente, o código inteiro (ASCII) correspondente ao caractere).

Vejamos o que acontece quando um programa trivial é executado.

```
#include <stdio.h>

main(){

    int ch;

    ch = getchar();
}
```

getchar() obtém sua entrada do teclado. Portanto, quando o programa acima é executado, o programa espera que o usuário digite alguma coisa. Cada caractere digitado é mostrado no monitor. O usuário pode digitar diversos caracteres na mesma linha, inclusive *backspace* para corrigir caracteres já digitados. No momento que ele teclar **RETURN** , o primeiro caractere da sequência digitada é o resultado da função getchar(). Portanto, na instrução do programa acima o caractere (ou melhor, o seu código ASCII) é atribuído a variável ch. Note que o usuário pode ter digitado diversos caracteres antes de teclar **RETURN** , mas a função getchar() só começará a ler o que foi digitado depois que for teclado **RETURN** . Além disso, com uma chamada da função getchar() só o primeiro caractere da sequência digitada é lida.

Você deve saber que o caractere de nova linha, `'\n'`, que tem o código ASCII 10, é automaticamente adicionado na sequência de caracteres de entrada quando o **RETURN** é teclado. Isso não tem importância quando a função getchar() é chamada uma única vez, mas isto pode causar problemas quando ele é usado dentro de um laço.

No inicício de qualquer programa que usa getchar(), você deve incluir

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

Esta diretiva do pré-processador diz ao compilador para incluir informações sobre getchar() e EOF (mais sobre EOF mais tarde.)

Considere o seguinte programa:

```
#include <stdio.h>

main(){

   int ch;

      printf( "Entre com uma letra: " );
      ch = getchar();
      if( ch < 'A' || ch > 'z' )
         printf( "Voce nao teclou uma letra!" );
      else
         printf( "Voce teclou %c, e seu codigo ASCII e %d.\n", ch, ch );
}
```

Um exemplo da execução do programa:

```
Entre com uma letra: A
Voce teclou A, e seu codigo ASCII e 65.
```

No exemplo de execução acima o usuário teclou A e depois **RETURN** .
Outro exemplo de execução do programa:

```
Entre com uma letra: AbcD
Voce teclou A, e seu codigo  ASCII e 65.
```

Neste caso o usuário digitou quatro caracteres e depois teclou **RETURN** . Embora quatro caracteres tenham sido digitados, somente uma chamada a função getchar() foi feita pelo programa, portanto só um caractere foi lido.

O tipo do resultado da função getchar() é int e não char. O valor é o código ASCII do caractere lido.

13.5.2 Marcando o final da entrada

Frequentemente quando você está digitando a entrada para o programa, você quer dizer ao programa que você terminou de digitar o que queria. Em Turbo C++, digitando ^Z (segure a tecla de Ctrl e pressione Z) você diz ao programa que terminou a entrada do programa.

Isto envia um sinal a função getchar(). Quando isso ocorre, o valor de ch depois de executar ch = getchar(); será um valor especial chamado EOF (que significa *end of file* – final do arquivo).

Considere o seguinte programa exemplo que conta o número de caracteres digitados (incluindo o caractere de "próxima linha"):

```
#include <stdio.h>

main(){

   int total = 0, ch;

   /* Le o proximo caractere em ch e para quando encontrar final do arquivo */
   while( (ch = getchar()) != EOF ) {
      total++;
   }
   printf( "\n%d caracteres digitados\n", total );
}
```

Só para esclarecer: você deve teclar **RETURN** depois de entrar com o comando ^Z.

13.5.3 Lendo o teclado usando getch() e getche()

As duas funções getch() e getche() são funções de entrada de leitura de caracter existentes em certos compiladores C existentes em ambiente MS-Windows, onde estão definidas na biblioteca padrão conio. Em ambientes UNIX, elas são definidas geralmente em bibliotecas não-padrão (por exemplo, curses). Descreveremos aqui o seu uso em ambiente MS-Windows.

A função getch() lê um caractere cada vez que é chamado. Os caracteres digitados não são mostrados no monitor. Cada caractere é lido no momento que é digitado (não é necessário teclar **RETURN**), ou seja, o que é digitado não é colocado em um *buffer*.

```
#include <conio.h>

main()
{
   char ch;
```

```
    printf("Digite algum caracter: ");

    ch = getch();

    printf("\n A tecla pressionada eh %c.\n", ch);

}
```

O Resultado deste programa na tela é:

```
Digite algum caracter:
 A tecla pressionada eh A.
```

A função getche() lê o caracter do teclado e mostra o que foi digitado na tela. Ela não aceita argumentos e devolve o caracter lido:

```
#include <conio.h>

main()
{
    char ch;

    printf("Digite algum caracter: ");

    ch = getche();

    printf("\n A tecla pressionada eh %c.\n", ch);

}
```

O Resultado deste programa na tela é:

```
Digite algum caracter: A
 A tecla pressionada eh A.
```

13.5.4 Para evitar problemas com a entrada...

(Observação: nesta seção, espaços em branco são relevantes e são mostrados como ␣)

Quando você executa um programa, cada caractere que você digita é lido e considerado como parte do *fluxo de entrada*. Por exemplo, quando você usa getchar(), você deve teclar **RETURN** no final. Como mencionado anteriormente, o primeiro caractere digitado é lido pelo getchar(). Mas, o caractere de nova linha continua no fluxo de entrada (porque você teclou **RETURN**).

De qualquer forma, se você executar um getchar() depois de um scanf() ou de um getchar() você lerá o caractere de nova linha deixado no fluxo de entrada.

Da mesma forma, quando você usa scanf() para ler informações, ele somente lê o que é necessário. Se voce usar scanf() para ler um número inteiro e digitar `42␣␣` (seguido de **RETURN**), o scanf() lê `42`, mas deixa `␣␣` (e o caractere de nova linha do **RETURN**) no fluxo de entrada.

Outro caso "problemático" é quando o scanf() é usado num laço. Se você digitar um valor do tipo errado, o scanf() lerá o valor errado e a execução do laço continuará na sentença após o scanf(). Na próxima iteração do laço o scanf() vai tentar ler novamente, mas o "lixo" deixado da iteração anterior ainda

estará lá, e portanto a chamada corrente do scanf() também não dará certo. Este comportamento resultará num laço infinito (um laço que nunca termina), ou terminará e terá um resultado errado.

Há uma maneira simples de resolver este problema; toda vez que você usar getchar() (para ler um caracter só) ou scanf(), você deve ler todo o "lixo" restante até o caractere de nova linha. Colocando as seguinte linhas após chamadas a getchar() ou scanf() o problema é eliminado:

```
/* Pula o restante da linha */
while( getchar() != '\n' );
```

Note que isso não é necessário após todas as chamadas a getchar() ou scanf(). Só depois daquelas chamadas que precedem getchar() (ou scanf()), especialmente em um laço.

A função scanf() na realidade retorna um inteiro que é o número de itens (valores) lidos com sucesso. Você pode verificar se o scanf() funcionou testando se o valor retornado é igual ao número de especificadores de formato no primeiro argumento da função.

```
main(){

   int total = 0, num;

   while( total < 20 ){
      printf( "Total = %d\n", total );

      printf( "Entre com um numero: " );
      if( scanf("%d", &num) < 1 )
          /* Ignora o resto da linha */
          while( getchar() != '\n' );
      else
         total += num;
   }

   printf( "Final total = %d\n", total );
}
```

13.6 Usando while e for

Embora qualquer laço possa ser escrito usando while ou for, a escolha é baseada principalmente no estilo. Por exemplo, se o laço precisa de uma inicialização e um incremento, então o for geralmente é usado. No caso em que o número de repetições não é pré-determinado em geral usa-se o while.

Como o comando for:

```
for( inicializacao; teste; incremento )
  sentenca;
```

é equivalente a:

```
inicializacao;
while( teste ) {
   sentenca;
   incremento;
};
```

você pode escolher o que preferir, a princípio.

14 Tipo Enumerado

Em muitos programas, variáveis do tipo `int` são utilizadas não por suas propriedades numéricas, mas para representar uma escolha dentre um pequeno número de alternativas. Por exemplo:

```
int sexo;  /* masculino = 1
              feminino = 2 */
int cor;   /* vermelho = 1
              amarelo = 2
              verde = 3    */
```

A utilização de códigos para representar os valores que uma variável poderá assumir, certamente compromete a clareza da estrutura de dados do programa, tornando sua lógica obscura e inconsistente. Por exemplo:

```
cor = 3;
if( sexo == 2 ) ...
cor = cor + sexo;
for( cor = 1; cor < 10; cor ++ )...
```

Um tipo enumerado permite definir uma lista de valores que uma variável deste tipo poderá assumir. A definição de um tipo enumerado é feita da seguinte forma:

$$\texttt{enum}\ Nome\_do\_tipo\ \{\ valor_1, valor_2, \ldots, valor_n\ \};$$

Exemplos de definição de tipos enumerados:

```
enum TpCores {VERMELHO, AMARELO, VERDE};
enum TpDias {SEG, TER, QUA, QUI, SEX, SAB, DOM};
enum TpSexos {MASCULINO, FEMININO};
```

Variáveis destes tipos são definidas da seguinte forma:

```
enum TpCores var1, var2;
enum TpDias  var3;
```

Agora, é possível dar valores a estas variáveis, por exemplo:

```
var1 = AMARELO;
var3 = QUI;
```

é um erro usar valores não definidos na declaração do tipo. A expressão `var2 = AZUL;` causa erro de compilação.

Internamente, o compilador trata variáveis enumeradas como inteiros. Cada valor na lista de valores possíveis corresponde a um inteiro, começando com 0 (zero). Portanto, no exemplo `enum TpCores`, `VERMELHO` é armazenado como `0`, `AMARELO` é armazenado como `1`, e `VERDE` é armazenado como `2`.

Utilização de tipos enumerados

Variáveis de tipos enumerados são geralmente usados para clarificar a operação do programa. Considere o seguinte trecho de programa que codifica dias da semana como inteiros (sendo `sabado = 5` e `domingo = 6`) para verificar se o dia do pagamento cai no final de semana e altera a dia para a segunda-feira seguinte.

```
#include <stdio.h>

/* prototipo da funcao que dada a data, retorna o dia da semana.
   seg=0, ter=1, qua=2, qui=3, sex=4, sab=5, dom=6 */

int diaDaSemana( int dia, int mes, int ano );

main(){
   int diaPgto, mesPgto, anoPgto;
   int diaSem;

   printf("Entre com a data de pagamento (dd mm aa): ");
   scanf("%d %d %d", &diaPgto, &mesPgto, &anoPgto);
   diaSem = diaDaSemana( diaPgto, mesPgto, anoPgto );
   if( diaSem == 5 )
     diaPgto = diaPgto + 2;
   else if( diaSem == 6 )
     diaPgto++;
   printf("Data do pagamento: %d/%d/%d\n", diaPgto, mesPgto,
   anoPgto);
}
```

Este programa ficaria mais legível se ao invés de codificar os dias da semana como inteiros e colocar a codificação como comentário, utilizar tipos enumerados. O programa ficaria então

```
#include <stdio.h>

enum TpSemana {SEG, TER, QUA, QUI, SEX, SAB, DOM};

/* prototipo da funcao que dada a data, retorna o dia da semana */
enum TpSemana  diaDaSemana( int dia, int mes, int ano );

main(){
   int diaPgto, mesPgto, anoPgto;
   int diaSem;

   printf("Entre com a data de pagamento (dd mm aa): ");
   scanf("%d %d %d", &diaPgto, &mesPgto, &anoPgto);
   diaSem = diaDaSemana( diaPgto, mesPgto, anoPgto );
   if( diaSem == SAB )
     diaPgto = diaPgto + 2;
   else if( diaSem == DOM )
     diaPgto++;
   printf("Data do pagamento: %d/%d/%d\n", diaPgto, mesPgto, anoPgto);
}
```

Note que a função `diaDaSemana` agora retorna apenas um dos valores da lista `SEG, TER, QUA, QUI, SEX, SAB, DOM` e portanto, no programa principal ao invés de testar se o `diaSem == 5` podemos escrever `diaSem == SAB`, o que torna o programa muito mais legível.

15 Arrays

Considere o seguinte programa. Este programa pede ao usuário notas de 4 estudantes, calcula a média e imprime as notas e a média.

```
main()
{
   int nota0, nota1, nota2, nota3;
   int  media;

   printf("Entre a nota do estudante 0: ");
   scanf("%d", &nota0);
   printf("Entre a nota do estudante 1: ");
   scanf("%d", &nota1);
   printf("Entre a nota do estudante 2: ");
   scanf("%d", &nota2);
   printf("Entre a nota do estudante 3: ");
   scanf("%d", &nota3);

   media = (nota0 + nota1 + nota2 + nota3) / 4;

   printf("Notas:  %d %d %d %d\n", gr0, gr1, gr2, gr3);
   printf("Media: %d\n", media);
}
```

Este programa é bem simples, mas ele tem um problema. O que acontece se o número de estudantes aumentar ? O programa ficaria muito maior (e feio !!). Imagine o mesmo programa se existissem 100 estudantes.

O que precisamos é uma abstração de dados para agrupar dados relacionados. Este é o objetivo de *arrays* em C.

Um *array* é uma coleção de um ou mais objetos, do mesmo tipo, armazenados em endereços adjacentes de memória. Cada objeto é chamado de *elemento* do array. Da mesma forma que para variáveis simples, damos um nome ao array. O tamanho do array é o seu número de elementos. Cada elemento do array é numerado, usando um inteiro chamado de *índice*. Em C, a numeração começa com `0` e aumenta de um em um. Assim, o último índice é igual ao número de elementos do array menos um.

Por exemplo, podemos definir um array `nota` de tamanho `100` para armazenar as notas dos cem estudantes:

```
int nota[100];
```

Quando o compilador encontra esta definição, ele aloca 200 bytes consecutivos de memória (dois bytes – referente a cada `int` – para cada nota). Cada nota pode ser acessada dando o nome do array e o índice entre colchetes: como `nota[0]` (para a primeira nota), `nota[1]` para a segunda nota, e assim por diantes, até a última nota, `nota[99]`.

15.1 Definindo arrays e acessando seus elementos

A definição de arrays é muito parecida com a definição de variáveis. A única diferença é que em array é necessário especificar seu tamanho (quantos elementos ele tem).

Os colchetes `[` e `]` são usados na definição do tamanho, como mostra os exemplos a seguir:

```
int total[5];

float tamanho[42];
```

O primeiro exemplo é um array de `5` inteiros (o tipo `int`) com o nome `total`. Como a numeração de arrays começa com `0`, os elementos da array são numerados `0`, `1`, `2`, `3` e `4`.

O segundo exemplo é um array de `42` elementos do tipo `float` com índices de `0` a `41`.

Cada elemento do array `total` é do tipo inteiro e pode ser usado do mesmo jeito que qualquer variável inteira. Para nos referirmos a um elemento do array, usamos colchetes também (`[` e `]`). O valor dentro dos colchetes pode ser qualquer expressão do tipo **inteiro**. Quando um array é definido, armazenamento suficiente (bytes contínuos na memória) são alocados para conter todos os elementos do array.

O nome do array representa um endereço de memória[3] constante que aponta para o início do espaço de armazenamento (o primeiro byte do bloco de bytes). O array, como um todo, não tem um valor, mas cada elemento individual tem um valor.

Note na tabela de precedência abaixo que `[ ]` tem precedência maior que todos os demais operadores.

Operador	Associatividade
`()  []  ->  .`	esquerda para direita
`!  -  ++  --  *  \&`	direita para esquerda
`*  /  %`	esquerda para direita
`+  -`	esquerda para direita
`<  <=  >  >=`	esquerda para direita
`==  !=`	esquerda para direita
`&&`	esquerda para direita
`||`	esquerda para direita
`=  +=  -=  *=  /=  %=`	direita para esquerda
`,`	esquerda para direita

Verifique se você entende as sentenças do programa abaixo.

```
int i, x, sala, total[5];
float area;
float tamanho[42];

x = total[3];

i = 4;

total[i] = total[i-1] + total[i-2];

total[4]++;

tamanho[17] = 2.71828;

sala = 3;

area = tamanho[sala] * tamanho[sala];

scanf("%f", &tamanho[41]);
```

Agora, podermos reescrever o programa que calcula a média de uma classe de 4 alunos:

```
main()
```

[3]Este valor de endereço de memória é conhecido em C como ponteiro. Este conceito será visto com detalhes nos próximos capítulos.

```
{
   int indice, nota[4], total = 0;

   for (indice = 0; indice < 4; indice++) {
      printf("Entre a nota do estudante %d: ", indice);
      scanf("%d", &nota[indice]);
   }

   printf("Notas:  ");
   for (indice = 0; indice < 4; indice++) {
      printf("%d ", nota[indice]);
      total += nota[indice];
   }
   printf("\nMedia: %d\n", total/4 );
}
```

Sample output:

```
Entre a nota do estudante 0: 93
Entre a nota do estudante 1: 85
Entre a nota do estudante 2: 74
Entre a nota do estudante 3: 100
Notas:  93 85 74 100
Media: 88
```

O programa é consideravelmente mais curto. Note que um `&` ainda precede o elemento do array passado para o `scanf()`. Não é necessário usar parênteses porque `[]` tem maior precedência que `&`.

O único problema é que ainda não é fácil modificar o programa para cem alunos porque `4` está em vários pontos do programa. Nós podemos usar o `#define` para manter o tamanho do array como uma constante simbólica ao invés de utilizar uma constante numérica.

```
#define ESTUDANTES 4

main()
{
   int nota[ESTUDANTES], indice, total = 0;


   for (indice = 0; indice < ESTUDANTES; indice++) {
      printf("Entre a nota do estudante %d: ",indice);
      scanf("%d", &nota[indice]);
   }
   printf("Grades:  ");
   for (indice = 0; indice < ESTUDANTES; indice++) {
      printf("%d ", nota[indice]);
      total += nota[indice];
   }
   printf("\nAverage: %d\n", total / ESTUDANTES );
}
```

15.2 Inicialização de arrays

Os arrays podem ser inicializados quando são definidos. Se o array não for inicializado, então ele contem "lixo".

Para inicializar um array, um valor para cada elemento deve ser especificado. Estes valores devem estar entre chaves (`{` e `}`) e são separados por vírgula (`,`). Alguns exemplos:

```
int valor[4] = { 1, 42, -13, 273 };

/* o tamanho do array pode ser omitido */
int peso[] = { 153, 135, 170 };
```

No primeiro exemplo, `valor` é um array de 4 inteiros onde `valor[0]` e' `1`, `valor[1]` e' `42`, `valor[2]` e' `-13`, e `valor[3]` e' `273`.

Note que no segundo exemplo, o tamanho do array foi omitido. Neste caso, o compilador calcula o tamanho como sendo o número de elementos listados. Quando um array é definido, se ele não for inicializado, o tamanho do array deve ser especificado. Se o array for inicializado, o tamanho pode ser omitido. O segundo exemplo acima é equivalente a

```
int peso[3] = { 153, 135, 170 };
```

Se o tamanho não for omitido, o número de elementos presentes não deve exceder o tamanho. Se exceder, o compilador gerará uma mensagem de erro. Se houver menos elementos na lista de inicialização, então os elementos dados são usados para inicializar os primeiros elementos do array. Qualquer elemento não inicializado conterá lixo.

Note que este tipo de inicialização só é válido no contexto onde o array é definido. Uma sentença como a seguinte produzirá um erro do compilador, uma vez que arrays só podem ser inicializados quando definidos.

```
int erro[5];

erro = { 2, 4, 6, 8, 10 };      /* ISTO ESTA' ERRADO */
```

Há mais uma restrição na inicialização de um array. Os valores devem ser todos constantes – nenhuma variável ou expressão é permitida. O seguinte trecho de programa produz um erro porque um dos valores de inicialização é uma variável:

```
int x = 21;
int yy[3] = { 1, 2, x };   /* ISTO ESTA' ERRADO */
```

15.3 Verificação de Limite

Quando um array é definido, é alocado espaço em memória para conter todos os elementos do array (não mais). O tamanho do array é dado explicitamente escrevendo o tamanho, ou implicitamente, inicializando o array. Embora arrays tenham tamanhos específicos, é possível que um programa tente acessar endereços de memória de elementos fictícios, ou seja, endereços de memória que não pertencem ao array. Isto acontece quando usamos um índice que não esteja entre `0` e `n-1` para um array de tamanho `n`. O compilador não gera nenhum aviso quando isto acontece. Quando executamos um acesso "fora dos limites" do array, o resultado pode ser desastroso. Isto siginifica que o programa pode não fazer nada, cancelar a execução, travar o computador, entrar em um loop infinito, etc.

Se você executar uma atribuição a um elemento do array fora do seu limite, você estará escrevendo em um endereço de memória que pode conter algo importante, destruindo-o. Em geral, erros como estes são difíceis de encontrar, já que o programa pode até executar, só que faz algo "estranho". Se você estiver usando o Turbo C++, você poderá ver uma mensagem como "Esta aplicação violou a integridade do sistema

devido a execução de uma instrução inválida e será cancelada.". Não entre em pânico !! Você terá que reinicializar o seu computador e examinar o seu programa cuidadosamente para achar acessos a array fora do seu limite. é claro que ao reinicializar o seu computador, você perderá todo o seu trabalho se não tiver salvado antes. **MORAL:** depois que seu programa compilar com sucesso, salve o seu programa em disco antes de executá-lo.

Por exemplo, considere o seguinte programa:

```
main()
{
    int ops[10], i;

    /* Acesso fora dos limites quando i = 10 */
    for (i = 0; i <= 10; i++)
        ops[i] = 0;
}
```

Este programa conta de `0` a `10`, inicializando cada elemento do array com `0`. O problema ocorre quando `i` tem o valor `10`. Neste ponto, o programa coloca `0` em `ops[10]`. Isto pode produzir resultados indefinidos (e desastrosos) embora o compilador não gere nenhum erro.

15.4 Arrays como argumentos

Para passar um array como argumento (com todos os seus elementos) passamos o nome do array. Considere o exemplo abaixo.

```
#define TAMANHO 5

int array_max(int []);

main()
{
   int i, valor[TAMANHO];

   for (i = 0; i < TAMANHO; i++) {
       printf("Entre um inteiro: ");
       scanf("%d", &valor[i]);
   }
   printf("O maior e' %d\n", array_max(valor));
}

/* funcao array_max(a)
 *  acao:      acha o maior inteiro de um array de TAMANHO elementos
 *  entrada:   array a de inteiros
 *  saida:     o maior valor do array
 *  suposicoes: a tem TAMANHO elementos
 *  algoritmo:  inicializa max com o primeiro elemento do array; em
 *              uma repeticao compara o max com todos os elementos
 *              do array em ordem e muda o valor de max quando um
 *              elemento do array for maior que o max ja' encontrado.
 */
int array_max(int a[])
```

```
{
   int i, max;

   /* Achar o maior valor do array */
   max = a[0];
   for (i = 1; i < TAMANHO; i++) {
       if (max < a[i])
          max = a[i];
   }
   return max;
}
```

Aqui está um exemplo de execução deste programa

```
Entre um inteiro: 73
Entre um inteiro: 85
Entre um inteiro: 42
Entre um inteiro: -103
Entre um inteiro: 15
O maior e' 85
```

Em `main()` a chamada para `array_max()` tem `valor` como seu argumento, que é copiado para o parâmetro formal `a`, que é um array de inteiros. Note que o tamanho não foi especificado, somente o nome do array, `a`. Porém é também correto incluir o tamanho (isto é uma questão de estilo – escolha o que você preferir):

```
int array_max(int a[TAMANHO])
{
  ...
}
```

A inclusão do tamanho de um array unidimensional na definição da função é somente por razões de legibilidade (para arrays multi-dimensionais, todas as dimensões exceto a primeira deve ser especificada).

Até este ponto, parece que não há diferença entre passar uma variável simples e um array como argumento para uma função. Mas há uma diferença fundamental: **QUANDO PASSAMOS O ARRAY COMO ARGUMENTO, ALTERAÇÕES NO ARRAY FEITAS DENTRO DA FUNÇÃO ALTERAM O CONTEÚDO DO ARRAY PASSADO COMO PARÂMETRO REAL**.

Note que isto não acontece quando uma variável simples é passada como argumento. Considere o exemplo seguinte:

```
void troca_a( int a );
void troca_vet( int vet[] );

main(){
   int x = 10;
   int v[] = {30, 40, 50};

   troca_a( x );
   printf("x=%d \n ", x);
   troca_vet( v );
   printf( "v[0]=%d v[1]=%d v[2]=%d", v[0], v[1], v[2] );
}
```

```
void troca_a( int a ){
   a = 20;
}

void troca_vet( int vet[] ){
   vet[0] = 60;
   vet[1] = 70;
   vet[2] = 80;
}
```

A saída deste programa é:

```
x=10
v[0]=60 v[1]=70 v[2]=80
```

O valor da variável `x` do programa principal não se altera porque como já vimos nas notas de aula 7, quando a função `troca_a` é chamada, o valor do argumento real `x` é avaliado, que é `10`, este valor é copiado para o parâmetro formal `a` da função `troca_a` e a função então é executada. O parâmetro `a` da função é tratada como variável local, portanto quando atribuímos `20` a `a`, estamos atribuindo `20` a uma variável local. Terminada a função, a execução retorna ao programa principal, que imprime o valor de `x`, que não foi alterado, ou seja, imprime `x=10`.

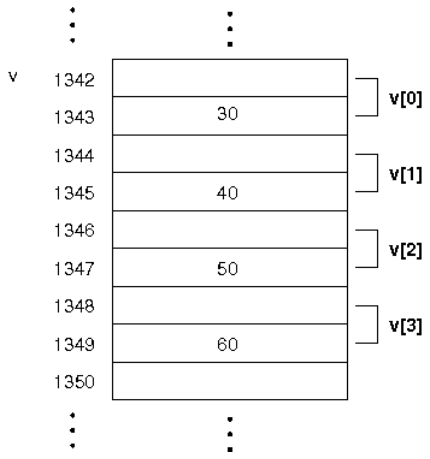
Quando a função `troca_vet` é chamada, o array `v` é passado como argumento e "copiado" para o parâmetro formal `vet`. A função é então executada, e os elementos do array são alterados para `60, 70, 80`. Como mencionado anteriormente, quando passamos um array como parâmetro, as alterações feitas no array dentro da função alteram o array passado como parâmetro. Portanto, quando a função termina e a execução continua no programa principal com a impressão dos valores dos elementos de `v`, será impresso `60, 70, 80`, os novos valores alterados de dentro da função `troca_vet`.

Vamos entender por que quando passamos só o nome do array como argumento as alterações afetam o array passado como parâmetro real. Como já mencionamos anteriormente, quando um array é definido, como `v` no programa principal acima, é alocado espaço suficiente na memória para conter todos os elementos do array. Na ilustração abaixo, são alocados 6 bytes de memória a partir do endereço `1342` para conter o array. O array como um todo não tem um valor, mas cada elemento do array tem (neste caso, foram inicializados com `30, 40, 60`). O nome do array, na verdade, contém o endereço onde começa o array, neste caso, o endereço `1342`.

Portanto, quando passamos o nome do array como argumento para uma função estamos na realidade passando como argumento o endereço de memória onde começa o array. No exemplo anterior, `1342` é passado como argumento para o parâmetro formal `vet` da função `troca_vet`. Portanto, da mesma forma que no caso da variável simples, o valor de `v`, que é o endereço `1342`, é copiado para o parâmetro `vet` de `troca_vet`. Então, quando a função `troca_vet` é executada, `vet` é um array de elementos do tipo `int` que começa no endereço `1342`. Quando atribuímos o valor `60` a `vet[0]`, estamos atribuindo `60` ao primeiro elemento do array que começa no endereço `1342`. Como este é o mesmo endereço onde começa o array `v` do programa principal, quando a função `troca_vet` termina, o array `v` "enxergará" o valor dos elementos do array que começa no endereço `1342`, que foram alterados pela função.

Quando passamos variáveis simples como argumento para uma função estamos passando somente o valor da variável, portanto, de dentro da função não é possível saber qual o endereço da variável para poder alterá-la.

Lembre-se que o endereço só é passado para a função quando passamos o array **COMO UM TODO** (ou seja, o nome do array, sem ser indexado por um elemento). Se passarmos como argumento apenas um elemento do array, o comportamento é o mesmo que se passássemos uma variável simples. Ou seja, o nome do array indexado por um valor entre colchetes refere-se ao valor do elemento do array, enquanto o nome do array sozinho refere-se ao endereço onde começa o array. Assim, no programa abaixo:




```
void troca_elem( int elem );
void troca_vet( int vet[] );

main(){
   int v[] = {30, 40, 50};

   troca_elem( v[0] );
   printf("v[0]=%d \n ", v[0]);
   troca_vet( v );
   printf( "v[0]=%d v[1]=%d v[2]=%d", v[0], v[1], v[2] );
}

void troca_elem( int elem ){
   elem = 20;
}

void troca_vet( int vet[] ){
   vet[0] = 60;
   vet[1] = 70;
   vet[2] = 80;
}
```

A saída do programa é:

```
v[0]=30
v[0]=60 v[1]=70 v[2]=80
```

Outro exemplo: a função `inicializaArray` abaixo inicializa todos os elementos do array `valor` com um valor passado como argumento pelo programa principal.

```
   #define TAMANHO 30

   /* Prototype */
   void inicializaArray(int [], int);

   main()
   {
      int valor[TAMANHO];
```

```
    /* Inicializa todos os elementos do array com 42 */
    inicializaArray(valor, 42);

    /* O resto do programa principal */
      .
      .
      .
}

/* funcao inicializaArray(a, k)
 *  acao:         inicializa todos os elementos de a com k
 *  entrada:      array de inteiros a, inteiro k
 *  saida:        nenhum
 *  suposicoes:   a tem TAMANHO elementos
 *  algoritmo:    uma repeticao for, inicializando um elemento a
 *                cada repeticao
 */
void inicializaArray(int a[], k)
{
    int i;

    for (i = 0; i < TAMANHO; i++)
      a[i] = k;
}
```

Como as alterações feitas por `inicializaArray` são vistas do programa principal, depois da função `inicializaArray` ser executada, no programa principal todos os elementos do array `valor` terão o valor `42`.

15.5 Exemplo: pesquisa linear de um array

Pesquisar (procurar) em um array um determinado valor (chamado de *chave*) é um problema muito comum em programação. Ele tem diversas aplicações. Por exemplo, podemos pesquisar um array de notas para verificar se algum aluno tirou `100` na prova. Há diversos algoritmos de pesquisa: cada um com suas vantagens e desvantagens. Nestas notas de aula, discutiremos um algoritmo simples, chamado de *pesquisa linear*. A pesquisa é feita usando uma repetição e examinando cada elemento do array a cada repetição e comparando o elemento com a chave que buscamos. A pesquisa termina quando um elemento do array que "casa" com a chave é encontrada, ou quando o array todo é percorrido e a chave procurada não é encontrada.

15.5.1 O Problema

Escreva uma função `pesquisa_linear` que tem como argumento de entrada: um array de inteiros a ser pesquisado, o tamanho do array, e uma chave (um valor inteiro) a ser procurado. A função retorna um inteiro: o índice do elemento do array (se a chave for achada) ou `-1` caso contrário.

1. Protótipo:

   ```
   int pesquisa_linear(int [], int, int);
   ```

2. Definição:

```
#define NAO_ACHOU -1

/* Procura uma chave em um array
 * entrada: array a ser pesquisado (arr ), tamanho do array (tam),
 *          chave a ser procurada (chave)
 * saida: o indice do elemento que e' igual a chave ou -1 caso nao ache
 * suposicao: nao assume que o array esteja ordenado
 */
int pesquisa_linear(int arr[], int tam, int chave)
{
  int i;

  for (i = tamanho - 1; i >= 0; i--) {
      if (arr[i] == chave)
         return i;
      }
  return NAO_ACHOU;
}
```

15.6 Exemplo: somar os elementos de dois arrays

15.6.1 O Problema

Escrever uma função que some dois arrays de `floats`, do mesmo tamanho. Dar o resultado em um terceiro array.

1. Protótipo:

```
void soma_array( float [], float [], float [], int );
```

2. Definição de `soma_array()`:

```
void soma_array( float arr1[], float arr2[], float arr3[], int tam )
{
   int i;

   for (i = 0; i < tam; i++)
       arr3[i] = arr1[i] + arr2[i];
}
```

15.7 Exemplo: ordenar um array

Um outro programa muito popular com arrays é ordená-lo de acordo com algum critério. Por exemplo, um array de inteiros pode ser ordenado em ordem crescente ou decrescente. O algoritmo chamado *Bubble sort* é bastante simples (porém não muito eficiente) de ordenação.

Basicamente, o algoritmo funciona da seguinte forma:

- na primeira passagem sobre o array: começando do último elemento do array até o segundo elemento, compare o valor de cada elemento com o valor do elemento anterior a ele. Se os elementos comparados estiverem fora de ordem, trocar os seus valores. Depois que esta primeira passada terminar, o que acontece é que o menor elemento do array torna-se o primeiro elemento do array.

- na segunda passagem pelo array: começando com o último elemento do array até o terceiro elemento, compare o valor de cada elemento com o valor do elemento anterior a ele. Se os dois elementos comparados estiverem fora de ordem, trocar os seus valores. Depois que esta passagem sobre o array terminar, o segundo menor elemento do array será o segundo elemento do array.
- repetir a passagem sobre o array de maneira similar até que a última passagem será simplesmente uma comparação dos valores do último elemento com o elemento anterior.

Por exemplo, se começarmos com um array: `9 8 7 6 5 4 3 2 1`, (o primeiro elemento é `9` e o último elemento é `1`) isto é o que aocntece com os elementos do array depois de cada passagem sobre ele (e troca de valores adjacentes):

```
passagem   conteudo do array depois da passagem
~~~~       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 -->       1  9  8  7  6  5  4  3  2

2 -->       1  2  9  8  7  6  5  4  3

3 -->       1  2  3  9  8  7  6  5  4

4 -->       1  2  3  4  9  8  7  6  5

5 -->       1  2  3  4  5  9  8  7  6

6 -->       1  2  3  4  5  6  9  8  7

7 -->       1  2  3  4  5  6  7  9  8

8 -->       1  2  3  4  5  6  7  8  9
```

Note que mesmo que se começássemos com um array ordenado de 9 elementos, ainda assim o algoritmo dado faz 8 passagens sobre o array. Para melhorar isso, nós apresentamos uma versão melhorada do *Bubble sort*.

A idéia é a seguinte. Antes de começar cada passagem, inicializamos uma variável `ordenado` com 1. Se durante a passagem uma troca de valores ocorrer, trocamos o valor da variável para 0. Assim, se depois da passagem, o valor da variável continuar sendo 1, isso significa que nenhuma troca ocorreu e que o array está ordenado.

15.7.1 Bubble sort

15.7.2 Algoritmo

Enquanto o array nao estiver ordenado

1. inicializar `ordenado` com 1
2. comparar pares adjacentes do array
   troque seus valores se estiver fora de ordem
   `ordenado` = 0.

15.7.3 Protótipo da função e definição

1. Protótipo

```
void bubble_sort(int [], int);
```

2. Definicao

```
/* Uma funcao que ordena um array de inteiros usando o algoritmo de
 * Bubble sort.
 * Entrada:  array a ser ordenado -- lista[]
 *           tamanho do array -- tam
 */
void bubble_sort(int lista[], int tam)
{
  int ordenado,       /* se 1 depois da passagem o array esta' ordenado */
      elem_final = 1, /* em uma passagem, elementos do ultimo ate' elem_fina
                         sao comparados com o elemento anterior */
      i,j,
      temp;

  /* enquanto o array nao estiver ordenado, fazer uma passagem sobre ele */
  do {
     ordenado = 1;  /* assume que array esta' ordenado */

     /* Examina o array do ultimo elemento ate elem_final.  Compara
        cada elemento com o anteior e troca seus valores se estiver
        fora de ordem.  */

     for (i = tam - 1; i >= elem_final; i--) {

         if (lista[i] < lista[i - 1]) {  /* troca os elementos de i e i-1 */
            temp = lista[i];
            lista[i] = lista[i - 1];
            lista[i - 1] = temp;
            ordenado = 0;         /* marca como nao ordenado */
         }
     }

     elem_final++;

     } while (!ordenado);
}
```

15.8 Comentários Finais

Neste curso, um dos únicos lugares que veremos o nome do array sem estar indexado é quando passamos o array (como um todo) para uma função. Para outras finalidades, veremos sempre o array indexado. Por exemplo, o seguinte trecho de programa está errado:

```
main(){
  int arr1[4] = {10, 20, 30, 40};
  int arr2[4];

  arr2 = arr1;       /* ERRADO: copia arr1 em arr2 */
```

```
                        /* tem que copiar elemento por elemento */

    if( arr1 == arr2 ) /* ERRADO: nao podemos comparar arrays inteiros */
      printf(``X'');   /* tem que comparar elemento por elemento */
}
```

16 Arrays Multidimensionais

Nas notas de aula anterior, apresentamos arrays unidimensionais. Em C, é possível também definir arrays de dimensões maiores. Eles são arrays de arrays. Um array de duas dimensões podem ser imaginado como uma matriz (ou uma tabela).

Como você deve ter imaginado, para definir e acessar arrays de dimensões maiores, usamos colchetes adicionais (`[` e `]`). Por exemplo:

```
int tabela[3][5];
```

Define um array bidimensional chamado `tabela` que é uma matriz 3 por 5 de valores do tipo `int` (15 valores no total). Os índices da primeira dimensão vão de 0 a 2, e os índices da segunda dimensão vão de 0 a 4.

Abaixo apresentamos um programa que imprime os elementos de um array bidimensional.

```
#include <stdio.h>
#define ALTURA 5
#define LARGURA 5

main()
{
   int x;                              /* numero da coluna */
   int y;                              /* numero da linha */
   char matriz [ALTURA] [LARGURA];     /* array 2-D [num_lins, num_cols] */


   /* preenche a matriz com pontos  */

   for(y=0; y<ALTURA; y++)
     for(x=0; x<LARGURA; x++)
        matriz[y][x] = '.';

   /* Imprime a matriz com pontos e a coordenada escolhida com X */

   printf("\nEntre coordenadas na forma y,x (2,4).\n");
   printf("Use valores negativos para sair do programa.\n");

   while (1){
      printf("Coordenadas: ");
      scanf("%d,%d", &y, &x);
      if (x >= 0 && y >= 0){
         matriz[y][x]='X';           /* coloca X no elemento escolhido */
         for(y=0; y<ALTURA; y++){    /* imprime o array todo */
            for(x=0; x<LARGURA; x++)
                printf("%c ", matriz[y][x] );
            printf("\n\n");
         }
         printf("\n");
      }
      else
         break;
```

```
    }
}
```

Neste exemplo, `matriz` é um array bidimensional. Ela tem número de elementos igual a `ALTURAxLARGURA`, sendo cada elemento do tipo `int`.

O exemplo abaixo preenche os elementos de um array bidimensional com os valores que representam a taboada e imprime a matriz.

```
/* Exemplo de array 2-D - taboada */

#include <stdio.h>
#define LIN 10
#define COL 10

main()
{
   int x;                          /* numero da coluna */
   int y;                          /* numero da linha  */
   int  tabela[LIN] [COL];          /* tabela de taboada  */

   /* preenche a tabela */

   for (y=0; y < LIN; y++)
       for(x=0; x < COL; x++)
           tabela[y][x] = y*x;

   printf("\n         Tabela de Multiplicacao\n");

   /* Imprime o numero das colunas */

   printf("%6d", 0);
   for (x=1; x < COL; x++)
       printf("%3d", x);
   printf("\n");

   /* Imprime uma linha horizontal */
   printf("   ");
   for (x=0; x < 3*COL; x++)
       printf("-");
   printf("\n");

   /* Imprime as linhas da tablea.
      Cada linha a precedida pelo indice de linha e uma barra vertical */

   for (y=0; y < LIN; y++) {
       printf("%2d|", y);
       for(x=0; x < COL; x++)
           printf("%3d", tabela[y][x]);
       printf("\n");
   }
```

```
}
```

A saída do programa é:

```
          Tabela de Multiplicacao
     0  1  2  3  4  5  6  7  8  9
   ----------------------------
 0|  0  0  0  0  0  0  0  0  0  0
 1|  0  1  2  3  4  5  6  7  8  9
 2|  0  2  4  6  8 10 12 14 16 18
 3|  0  3  6  9 12 15 18 21 24 27
 4|  0  4  8 12 16 20 24 28 32 36
 5|  0  5 10 15 20 25 30 35 40 45
 6|  0  6 12 18 24 30 36 42 48 54
 7|  0  7 14 21 28 35 42 49 56 63
 8|  0  8 16 24 32 40 48 56 64 72
 9|  0  9 18 27 36 45 54 63 72 81
```

16.1 Inicialização

Arrays multidimensionais podem ser inicializados usando listas aninhadas de elementos entre chaves. Por exemplo, um array bidimensional `tabela` com três linhas e duas colunas pode ser inicializado da seguinte forma:

```
double tabela[3][2] = { {1.0,  0.0},      /* linha 0 */
                        {-7.8, 1.3},      /* linha 1 */
                        {6.5,  0.0}       /* linha 2 */
                      };
```

Quando o array é inicializado, o tamanho da primeira dimensão pode ser omitido. A definição de array abaixo é equivalente a dada anteriormente.

```
double tabela[][2] = { {1.0,  0.0},      /* linha 0 */
                       {-7.8, 1.3},      /* linha 1 */
                       {6.5,  0.0}       /* linha 2 */
                     };
```

16.2 Arrays Multidimensionais – arrays de arrays

O formato da definição de um array de dimensão k, onde o número de elementos em cada dimensão é $n_0, n_1, \ldots, n_{k-1}$, respectivamente, é:

$$nome\_tipo\ nome\_array[n_0][n_1]...[n_{k-1}];$$

Isto define um array chamado $nome\_array$ consistindo de um total de $n_0 \times n_1 \times \ldots \times n_{k-1}$ elementos, sendo cada elemento do tipo $nome\_tipo$.

Arrays multidimensionais são armazenados de forma que o último subscrito varia mais rapidamente. Por exemplo, os elementos do array

```
int tabela[2][3];
```

são armazenados (em endereços consecutivos de memória) como

```
tabela[0][0], tabela[0][1], tabela[0][2], tabela[1][0], tabela[1][1], tabela[1][
```

Um array de dimensão k, onde o número de elementos em cada dimensão é $n_0, n_1, \ldots, \ldots, n_{k-1}$, respectivamente, pode ser imaginado como um array de dimensão n_0 cujos elementos são arrays de dimensão $k-1$.

Por exemplo, o array bidimensional `tabela`, com 20 elementos do tipo `int`

```
int tabela[4][5] = { {13, 15, 17, 19, 21},
                     {20, 22, 24, 26, 28},
                     {31, 33, 35, 37, 39},
                     {40, 42, 44, 46, 48} };
```

pode ser imaginado como um array unidimensional de 4 elementos do tipo `int[]`, ou seja, arrays de `int`; cada um dos 4 elementos é um array de 5 elementos do tipo `int`:

```
tabela[0]  ---> {13, 15, 17, 19, 21}
tabela[1]  ---> {20, 22, 24, 26, 28}
tabela[2]  ---> {31, 33, 35, 37, 39}
tabela[3]  ---> {40, 42, 44, 46, 48}
```

16.3 Arrays Multidimensionais como argumento para funções

Quando o parâmetro formal de uma função é um array multidimensional (um array com dimensão maior que um), todas as dimensões deste array, exceto a primeira, precisa ser explicitamente especificada no cabeçalho e protótipo da função.

$$tipo\_do\_resultado\ nome\_da\_funcao\,(\,nome\_do\_tipo\ nome\_do\_array[\,][n_1]...[n_{k-1}], ...\,)$$

Quando uma função com um parâmetro formal do tipo array é chamada, na chamada da função **somente** o nome do array é passado como parâmetro real. O tipo (e portanto a dimensão) do array passado como parâmetro real deve ser consistente com o tipo (e portanto a dimensão) do array que é o parâmetro formal. O programa abaixo mostra o exemplo da tabela de multiplicação escrita usando funções.

```
/* Exemplo de array 2-D - tabela de multiplicacao */

#include <stdio.h>
#define LIN 10
#define COL 10

void inicializa_arr (int arr[][COL], int);
void imprime_arr (int arr[][COL], int);

main()
{
   int  tabela[LIN] [COL];

   inicializa_arr(tabela, LIN);

   printf("\n         Tabela de Multiplicacao\n");

   imprime_arr(tabela, LIN);
}
```

```
/* Inicializa o array com a tabela de multiplicacao */

void inicializa_arr (int arr[][COL], int nLIN)
{
   int x;                          /* numero da coluna */
   int y;                          /* numero da linha */

   /* preenche o array */

   for (y=0; y < nlin; y++)
       for(x=0; x < COL; x++)
           arr[y][x] = y*x;
}

/* imprime um array LIN x COL */

void imprime_arr(int arr[][COL], int nlin)
{
   int x;                          /* numero da coluna */
   int y;                          /* numero da linha */

   /* imprime o numero das colunas */

   printf("%6d", 0);
   for (x=1; x < COL; x++)
       printf("%3d", x);
   printf("\n");


   /* imprime uma linha horizontal */
   printf("   ");
   for (x=0; x < 3*COL; x++)
       printf("_");
   printf("\n");

   /* imprime as linhas do array.  cada linha e' precedida
      pelo numero da linha e uma barra vertical */

   for (y=0; y < nlin; y++) {
       printf("%2d|", y);
       for(x=0; x < COL; x++)
           printf("%3d", arr[y][x]);
       printf("\n");
   }
}
```

Outro exemplo com funçoes de manipulação de arrays bidimensionais:

```
/* funcoes com argumentos tipo array 2-D */
#include <stdio.h>
```

```
#define ALTURA 5
#define LARGURA 5

void seleciona_elem    (int [][LARGURA], int);
void pontos       (int [][LARGURA], int);
void imprime_matriz  (int [][LARGURA], int);
void marca_triang (int [][LARGURA], int);
void flip       (int [][LARGURA], int);
void espera_entrada(void);

/**** DEFINICAO DE FUNCOES ********/

/* funcao que preenche uma matriz nlin X LARGURA com pontos */
void pontos( int matriz[][LARGURA], int nlin)
{
   int x,y;

   for(y=0; y<nlin; y++)
     for(x=0; x<LARGURA; x++)
        matriz[y][x] = '.';

}

/* funcao que preenche os elementos selecionados da matriz com um
 * quadrado e imprime a matriz
 */
void seleciona_elem(int matriz[][LARGURA], int nlin)
{
   int x, y;

   printf("\nEntre com as coordenadas na forma y,x (2,4).\n");
   printf("Use numeros negativos para terminar.\n");

   while (1)
      {
      printf("Coordenadas: ");
      scanf("%d,%d", &y, &x);
      if (x >= 0 && y >= 0)
         {
         matriz[y][x]='\xB1';            /* preenche o elemento com quadrado */
         imprime_matriz(matriz, nlin);   /* imprime a matriz */
         }
      else
         break;
      }
 }

/* funcao que marca todos os elementos abaixo da diagonal principal de
 * um array nlin X LARGURA com quadrados
 */
void marca_triang(int matriz[][LARGURA], int nlin)
```

```
{
  int x, y;

  printf("Triangulo\n");
  pontos(matriz, nlin);
  for (y = 0; y < nlin; y++)
    for (x = 0; x <= y; x++)
      matriz[y][x] = '\xB1';
}

/* funcao que imprime um array 2-D nlin X LARGURA */
void imprime_matriz(int matriz[][LARGURA], int nlin)
{
  int x,y;

  for(y=0; y<nlin; y++)
{
for(x=0; x<LARGURA; x++)
   printf("%c ", matriz[y][x] );
printf("\n\n");
}
  printf("\n");
}

/* funcao que flipa um array ao longo da diagonal principal */
void flip(int matriz[][LARGURA], int nlin)
{
  int x, y;
  int temp;

  printf("Flipado ao longo da diagonal principal.\n");
  for (y = 0; y < nlin; y++)
    for (x = 0; x <= y; x++){
      temp = matriz[y][x];
      matriz[y][x] = matriz[x][y];
      matriz[x][y] = temp;
    }
}

/* funcao que espera ate que uma tecla seja digitada */
void espera_entrada( void ) {
   getchar(); }

/********** MAIN ************/

/* alguns exemplos de chamadas de funcoes com argumentos array 2-D */
main()
{
   int matriz [ALTURA] [LARGURA];

   pontos(matriz, ALTURA);
```

```
    seleciona_elem(matriz, ALTURA);
    espera_entrada();

    flip(matriz, ALTURA);
    imprime_matriz(matriz,ALTURA);
    espera_entrada();

    marca_triang( matriz, ALTURA);
    imprime_matriz( matriz, ALTURA);
    espera_entrada();

    flip( matriz, ALTURA);
    imprime_matriz(matriz, ALTURA);
    espera_entrada();
}
```

17 Array de Caracteres

Nas notas de aula anteriores, enfatizamos arrays de números. Em geral, podemos ter arrays com elementos de qualquer um dos tipos vistos até agora (incluindo arrays – visto nas notas de aula 9). Nesta seção, apresentaresmo arrays com elementos do tipo `char`.

Abaixo, apresentamos um exemplo de programa que define e inicializa um array de caracteres, e depois imprime o array em ordem reversa.

```
main()
{
   char arr1[] = {'c','i','2','0','8'};
   int i;

   for (i = 4; i >= 0; i--)
       printf("%c", arr1[i]);
}
```

Arrays de caracteres são usados para armazenar texto, mas é muito inconveniente se tivermos que colocar cada caractere entre apóstrofes. A alternativa dada pela linguagem C é

```
   char arr2[] = "ci208" ;
```

Neste caso, "`ci208`" é um *string* de caracteres ou uma constante do tipo *string*. Nós já usamos *strings* antes, com as funções `printf()` e `scanf()` (constantes do tipo *string* estão sempre entre aspas - "):

```
   printf("Entre com a nota para o estudante 2: ");
   scanf("%d", &gr2);
```

18 Strings

Strings são arrays de caracteres (arrays com elementos do tipo `char`) que DEVEM terminar com `'\0'` (o caracter NUL). Se você usa o nome NUL em seu programa, então é necessária a definição `#define NUL '\0'` .

No exemplo acima, embora não tenhamos escrito explicitamente o caracter NUL, o compilador automaticamente o colocou como o último elemento do array `arr2[]`. Portanto, o tamanho de `arr2[]` é 6: 5 para os caracteres que digitamos (ci208) e 1 para o caractere NUL que o compilador introduziu automaticamente. As definições abaixo são equivalentes.

```
   char arr2[] =  {'c','i',' ', '2','0','8','\0'};

   char arr2[] =  {'c','i',' ', '2','0','8', NUL};
```

O caractere NUL marca o fim de um string. Outros exemplos:

```
/* a maneira tediosa */
char name1[] = { 'j','o','s','e',' ','s','i','l','v','a','\0' };

/* e a maneira facil */
char name2[] = "jose silva";
```

Embora o primeiro exemplo seja um string, o segundo exemplo mostra como strings são geralmente escritos (como constantes). Note que se você usar aspas quando escreve uma constante, você não precisa colocar `'\0'`, porque o compilador faz isso para você.

Quando você for criar um array de caracteres de um tamanho específico, lembre-se de adicionar 1 para o tamanho máximo de caracteres esperado para armazenar o caractere NUL. Por exemplo, para armazenar o string "`programar e divertido`", você precisa de um array de tamanho 22 (21 caracteres + 1 para o NUL).

18.1 Imprimindo strings com `puts()` e `printf()`

Strings podem ser impressos usando `printf()` com o especificador de formato `%s`. Por exemplo:

```
main()
{
    char mensagem[] = "tchau";

    printf("ola\n%s\n", mensagem);
}
```

A saída deste programa é:

```
ola
tchau
```

A função `puts()` simplesmente imprime um string e depois pula de linha. Nenhuma opção de formatação pode ser definida. A função `puts()` somente pega um string como argumento e o imprime. O programa abaixo tem a mesma saída que o programa anterior.

```
main()
{
    char mensagem[] = "tchau";

    puts("ola");
    puts(mensagem);
}
```

18.2 Lendo strings do teclado com `gets()` e `scanf()`

A função `gets()` lê uma linha de texto digitado no teclado e a armazena em um string. Veja o exemplo abaixo:

```
main()
{
    char nome[100];

    printf("Entre seu nome: ");
    gets(nome);
    printf("Oi, %s.\n", nome);
}
```

Exemplo de execução

```
Entre seu nome: Jose Silva
Oi, Jose Silva.
```

Passando um nome de array para a função `gets()`, como ilustrado no programa acima, coloca a linha inteira digitada pelo usuário no array `nome` (tudo até que seja teclado enter). Note que se o usuário digitar caracteres demais (neste caso, mais de 99 caracteres), isso causará um erro de acesso fora dos limites (que pode ser PERIGOSO !!)

A função `scanf()` pode ser usada de maneira similar. A única diferença é que o `scanf()` lê somente a primeira palavra (tudo até que de digite um separador – um espaço em branco, tabulação, ou enter). Além disso, como estamos passando um array como argumento para o `scanf()`, **O & QUE GERALMENTE PRECEDE O ARGUMENTO NÃO DEVE ESTAR PRESENTE**.

```
main()
{
    char nome[100];

    printf("Entre seu nome: ");
    scanf("%s", nome);
    printf("Oi, %s.\n", nome);
}
```

Exemplo de execução

```
Entre seu nome: Jose Silva
Oi, Jose.
```

Note que somente o primeiro nome é lido pelo `scanf()` porque a função para no primeiro espaço em branco que encontra (enquanto `gets()` para quando encontra um enter).

18.3 Array de Strings

Em notas de aula anteriores, vimos alguns exemplos de arrays de arrays (matrizes ou tabelas). Como strings são também arrays, podemos definir arrays de strings. O programa abaixo inicializa um array de strings com nomes e os imprime.

```
#define NUM_NOMES 5        /* define a quantidade de nomes no array */
#define TAM       20       /* define o tamanho maximo do nome */

main()
{
    char nomes[NUM_NOMES][TAM] = {"Jose Silva",
                                  "Maria Silva",
                                  "Antonio dos Santos",
                                  "Pedro dos Santos",
                                  "Joao da Silva"};
    int i;

    for(i = 0; i < 5; i++)
      printf("%s\n", nomes[i]);
}
```

A saída deste programa é:

```
Jose Silva
Maria Silva
Antonio dos Santos
Pedro dos Santos
Joao da Silva
```

18.4 Funções de String

Há funções para manipulação de string já definidas na biblioteca padrão C chamada `string.h`. Todas as funções que apresentaremos nesta seção são parte desta biblioteca. Portanto, se seu programa utilizar uma destas funções você deve incluir a linha `#include <string.h>` no início do seu programa.

O objetivo desta seção é mostrar como estas funções poderiam ser implementadas como exemplos de programas de manipulação de strings.

18.4.1 A função `strlen()`

A função `strlen()` tem como argumento um string. Ela retorna um inteiro que é o comprimento do string (o número de caracteres do string, não contando o caractere NUL). Por exemplo, o comprimento do string "`alo`" é `3`.

```
main()
{
    char nome[100];
    int comprimento;

    printf("Entre seu nome: ");
    gets(nome);
    comprimento = strlen(nome);

    printf("Seu nome tem  %d caracteres.\n", comprimento);
}
```

Um exemplo de execução:

```
Entre seu nome: Dostoevsky
Seu nome tem 10 caracteres.
```

Abaixo, mostramos como a função `strlen()` poderia ser implementada.

```
int strlen( char str[] )
{
   int comprimento = 0;


   while ( str[comprimento] != NUL )
       comprimento++;
   return comprimento;
 }
```

18.4.2 A função `strcmp()`

A função `strcmp()` é usada para comparar dois strings. Lembre que não podemos usar ==, como em `str1 == str2`, para comparar dois strings, uma vez que strings são arrays. Strings devem ser comparados caractere por caractere. A função `strcmp()` tem como argumento dois strings e retorna um inteiro.

Strings são ordenados de forma similar a maneira como palavras são ordenadas em um dicionário. Ordenamos palavras em um dicionário alfabeticamente, e ordenamos strings respeitando a ordem dos caracteres no conjunto de caracteres da máquina. A ordenação abaixo é válida em qualquer computador:

```
'0' < '1' < ... < '8' < '9'
'A' < 'B' < ... < 'Y' < 'Z'
'a' < 'b' < ... < 'y' < 'z'
```

A ordem relativa do três conjuntos (dígitos, letras maiúsculas e letras minúsculas) depende do computador utilizado.

Se `s1` e `s2` são strings, o resultado da chamada de função `strcmp(s1, s2)` é:

- se `s1` $=_s$ `s2`, `strcmp()` retorna `0`
- se `s1` $<_s$ `s2`, `strcmp()` retorna um número negativo (`< 0`)
- se `s1` $>_s$ `s2`, `strcmp()` retorna um inteiro positivo (`> 0`)

(onde $=_s$, $<_s$ e $>_s$ são $=$, $<$ e $>$ para strings)

$s1 <_s s2$ significa "$s1$ vem antes de $s2$ no dicionário". Exemplos: "`tudo`" é menor que "`xadrez`", "`calor`" é menor que "`calorao`", "`frio`" é menor que "`quente`", e é claro o string vazio , `NUL`, é menor que qualquer string.

Considere o exemplo abaixo que usa `strcmp()`:

```
main()
{
    char palavra1[100], palavra2[100];
    int resultado;

    printf("entre com uma palavra: ");
    gets(palavra1);
    printf("entre outra palavra: ");
    gets(palavra2);

    resultado = strcmp(palavra1, palavra2);

    if (resultado == 0) {
        printf("igual\n");
    } else if (resultado > 0) {
        printf("o primeiro e' maior\n");
    } else {
        printf("o segundo e' maior\n");
    }
}
```

Aqui está um exemplo de como a função `strcmp()` poderia ser implementada.

```
int strcmp( char s1[], char s2[] )
{
   int i = 0;

   while (1)
```

```
    {
    if (s1[i] == NUL && s2[i] == NUL)
       return 0;
    else if (s1[i] == NUL)
       return -1;
    else if (s2[i] == NUL)
       return 1;
    else if (s1[i] < s2[i])
       return -1;
    else if (s1[i] > s2[i])
       return 1;
    else
       {
       i++;
       }
    }
}
```

Na biblioteca padrão, a função `strcmp()` faz distinção entre letras maiúsculas e minúsculas. Se você não quer que a função faça esta distinção, você pode modificar o seu string para ter apenas letras minúsculas (ou maiúsculas) antes de passá-lo como argumento para a função `strcmp()`. Para fazer isso, você pode usar a função da biblioteca padrão `tolower()`, que tem como argumento um caractere. Se o caractere passado é uma letra maiúscula, ele retorna esta letra minúscula; caso contrário, retorna o mesmo caractere. Por exemplo: `tolower('A')` é `'a'`, `tolower('1')` é `'1'`, `tolower('a')` é `'a'`.

18.4.3 A função `strcpy()`

A função `strcpy()` é usada para copiar o conteúdo de um string para outro. Ela tem dois argumentos: `strcpy(s1, s2)` copia o conteúdo do string `s2` para o string `s1`. A função `strcpy()` que apresentamos abaixo não retorna um valor. Seu protótipo é

```
void strcmp(char [], char []);
```

O exemplo abaixo mostra a utilização do `strcpy()`.

```
main()
{
    char pal[100], palCopia[100];

    printf("entre com uma palavra: ");
    gets(pal);
    strcpy(palCopia, pal);

    printf("entre outra palavra: ");
    gets(pal);

    printf("voce entrou primeiro: %s\n", palCopia);
}
```

Embora este programa pudesse ter sido escrito sem usar `strcpy()`, o objetivo é mostrar que se pode usar `strcpy()` para fazer "atribuição" de strings.

A função `strcpy()` poderia ter sido implementada da seguinte forma:

```
void strcpy( char s1[], char s2[] )
{
   int i = 0;

   while ( s2[i] != NUL ) {
       s1[i] = s2[i];
       i++;
   }
   s1[i] = s2[i];
}
```

19 Estruturas

A estrutura de dados ***array*** é usada para conter dados do mesmo tipo junto. Dados de tipos *diferentes* também podem ser agregados em tipos chamados de **estruturas** ou **registros** (tipo `struct` em linguagem C). Primeiro, o tipo estrutura é declarado (precisamos especificar que tipos de variáveis serão combinados na estrutura), e então variáveis deste novo tipo podem ser definidas (de maneira similar que usamos para definir variáveis do tipo `int` ou `char`).

19.1 Declaração de Estruturas

Uma declaração de estrutura *declara* um tipo *struct*. Cada tipo *struct* recebe um *nome* (ou *tag*). Refere-se àquele tipo pelo *nome* precedido pela palavra `struct`. Cada unidade de dados na estrutura é chamada *membro* e possui um *nome de membro*. Os membros de uma estrutura podem ser de *qualquer* tipo. Declarações de estrutura não são definições. Não é alocada memória, simplesmente é introduzida um novo tipo de estrutura.

Geralmente declarações de estruturas são globais. Elas são colocadas próximas ao topo do arquivo com o código fonte do programa, assim elas são visíveis por todas as funções (embora isto dependa de como a estrutura está sendo usada).

A forma padrão de declaração de uma estrutura é:

`struct` *nome-estrutura* `{`
    declaração dos membros
`}` *definição de variáveis (optional)*;

Abaixo se apresenta um exemplo de um tipo estrutura que contém um membro do tipo `int` e um outro membro do tipo `char`.

```
struct facil {
    int num;
    char ch;
};
```

Esta declaracao cria um novo tipo chamado `struct facil` que contém um inteiro chamado `num` e um caracter chamado `ch`.

19.2 Definição de variáveis de um tipo estrutura declarado

Como acontece com qualquer outro tipo de dados, variáveis de tipos de estruturas são definidas fornecendo o nome do tipo e o nome da variável. Considere a definição abaixo relativa a uma variável com o nome `fac1` que é do tipo `struct facil`:

```
struct facil fac1;
```

Tal definição está associada com a alocação de memória: memória suficiente será alocada para guardar um `int` e um `char` (nesta ordem). Como qualquer outra variável, `fac1` tem um nome, um tipo, e um endereço associados.

Variáveis de estruturas possuem também valores, e como outras variáveis locais, se elas não tem atribuídas um valor específico, seu valor é indefinido.

É possível definir variáveis durente a declaração do tipo estrutura:

```
struct facil {
    int num;
    char ch;
} fac1;
```

Note-se que sem conflito, nomes de membros (tais como `num` e `ch`) podem ser usados como nomes de outras variáveis independentes (fora do tipo estrutura definido) or como nomes de membros em outros tipos estrutura. No entanto, deve-se evitar situações que criem confusão.

19.3 Acesso a membros de uma estrutura: ponto (.), o operador membro de estrutura

Dada uma variável de estrutura, um membro específico é referenciado usando o nome da variável seguida de `.` (ponto) e pelo nome do membro da estrutura. Assim, as seguintes referências a membros de uma estrutura são válidas:

`fac1.num` se refere ao membro com nome `num` na estrutura `fac1`;
`fac1.ch` se refere ao membro com nome `ch` na estrutura `fac1`.

Membros de estrutura (como `fac1.num`) são variáveis, e podem ser usadas como valores (no lado direito de uma atribuição, em expressões como argumentos para funções), ou como *lvalues* (no lado esquerdo de atribuições, com operadores de incremento/decremento ou com o operador de endereço (&)). O exemplo abaixo mostra alguns exemplos do uso de membros de estrutura:

```
fac1.ch = 'G';
fac1.num = 42;

fac1.num++;

if (fac1.ch == 'H') {
   printf("%d\n", fac1.num);
}
```

Tentar acessar um nome de membro que não existe causa um erro de compilação.

19.4 Operadores usados com variáveis de estrutura: valores e *lvalues*

Uma variável de estrutura pode ser tratada como um objeto simples no todo, com um valor específico associado a ela (a estrutura `fac1` tem um valor que agrega valores de todos os seus membros). Note a diferença com *arrays*: se `arr[]` é um array de tamanho 2 definedo como `int arr[2] = {0,1};`, o nome `arr2` não se refere ao valor coletivo de todos os elementos do *array*. Na verdade, `arr2` é um ponteiro constante e se refere ao endereço de memória onde o *array* se inicia. Além disso `arr2` não é um *lvalue* e não pode ser mudado. Variáveis de estrutura são diferentes. Elas podem ser usadas como valores e *lvalues*, mas com certas limitações.

Os únicos usos válidos de uma variável de estrutura são dos dois lados de um operador de atribuição (=), como operando do operador de endereço `&` (obtendo o endereço da estrutura), e referenciando seus membros.

De todas as variações de atribuição (incluindo o incremento e decremento) atribuição de estruturas pode ser usada APENAS com =. O uso de outros operadores de atribuição ou de incremento causará um erro de compilação A atribuição de um valor de estrutura para outro copia *todos* os membros de uma estrutura para outra. Mesmo que um dos membros seja um *array* ou outra estrutura, ela é copiada integralmente. As duas estruturas envolvidas na atribuição devem ser do mesto tipo `struct`. Considere o seguinte exemplo:

```
struct facil {
    int num;
    char ch;
};

main()
{
```

```
    struct facil fac1, fac2;

    fac1.num = 3;
    fac1.ch = 'C';

    /* Atribuindo fac1 a fac2 */
    fac2 = fac1;
}
```

Lembre-se que este tipo de atribuição é ilegal com *arrays*. Tentar fazer isto com dois *arrays* causa um erro de compilação (uma vez que nomes de *arrays* são ponteiros constantes).

```
int a[5], b[5];

/* Está errado -- Não irá compilar */
a = b;
```

19.5 Inicialização de estruturas

Variáveis de estruturas não-inicializadas contém valores indefinidos em cada um de seus membros. Como em outras variáveis, variáveis de estruturas podem ser inicializadas ao serem declaradas. Esta inicialização é análoga ao que é feito no caso de *arrays*. O exemplo abaixo ilustra a inicialização de estruturas:

```
struct facil {
    int num;
    char ch;
};

main()
{
    struct facil fac1 = { 3, 'C' }, fac2;

    fac2 = fac1;
}
```

Uma lista de valores separados por vírgula fica entre chaves (`{` and `}`). Os valores de inicialização devem estar na mesma ordem dos membros na declaração da estrutura.

19.6 Estruturas como argumentos de função e valores de retorno

Como qualquer outro valor do tipo `int` ou `float`, valores de estruturas podem ser passados como argumentos para funções, e podem ser retornados de funções. O exemplo abaixo ilustra tal prorpiedade:

```
#define LEN 50

struct endereco {
    char rua[LEN];
    char cidade_estado_cep[LEN];
};

struct endereco obtem_endereco(void);
void imprime_endereco(struct endereco);
```

```
struct endereco obtem_endereco(void)
{
     struct endereco ender;

     printf("\t Entre rua: ");
     gets(ender.rua);
     printf("\t Entre cidade/estado/cep: ");
     gets(ender.cidade_estado_cep);

     return ender;
}

void imprime_endereco(struct endereco ender)
{
     printf("\t %s\n", ender.rua);
     printf("\t %s\n", ender.cidade_estado_cep);
}

main()
{
     struct endereco residencia;

     printf("Entre seu endereco residencial:\n");
     residencia = obtem_endereco();

     printf("\nSeu endereco eh:\n");
     imprime_endereco(residencia);
}
```

No exemplo acima, a estrutura `struct endereco` contém dois *arrays* de tamanho 50. Dentro da função `obtem_endereco()`, a variável `ender` é declarada como sendo do tipo `struct endereco`. Após usar `gets()` para o fornecimento da informação, o valor de `ender` é retornado para `main()`, de onde a função `obtem_endereco()` foi chamada. Este valor é então passado para a função `imprime_endereco()`, onde o valor de cada membro da estrutura é exibido na tela.

Este programa pode ser comparado ao programa abaixo, que usa valores do tipo `int` no lugar de valores do tipo `struct endereco` (claro que a informação lida e exibida é um simples valor numérico, e não um nome de rua, etc.):

```
int obtem_int(void);
void imprime_int(int);

int obtem_int(void)
{
     int i;

     printf("Entre valor: ");
     scanf("%d", &i);

     return i;
}
```

```
void imprime_int(int i)
{
     printf("%d\n", i);
}

main()
{
     int valor;

     valor = obtem_int();

     printf("\nSeu valor:\n");
     imprime_int(valor);
}
```

19.7 Arrays de estruturas

Arrays de estruturas são como *arrays* de qualquer outro tipo. Eles são referenciados e definidos da mesma forma. O exemplo abaixo é análogo ao exemplo de endereço apresentado anteriormente, exceto que uma quantidade de `NUM` endereços é armazenada ao invés de apenas um.

```
#define LEN 50
#define NUM 10

struct endereco {
    char rua[LEN];
    char cidade_estado_cep[LEN];
};

void obtem_endereco(struct endereco [], int);
void imprime_endereco(struct endereco);

void obtem_endereco(struct endereco aender [], int index)
{
     printf("Entre rua: ");
     gets(aender[index].rua);
     printf("Entre cidade/estado/cep: ");
     gets(aender[index].cidade_estado_cep);
}

void imprime_endereco(struct endereco ender)
{
     printf("%s\n", ender.rua);
     printf("%s\n", ender.cidade_estado_cep);
}

main()
{
     struct endereco residencias[NUM];
     int i;
```

```
    for (i = 0; i < NUM; i++) {
        printf("Entre o endereco da pessoa %d:\n", i);
        obtem_endereco(residencias,i);
    }

    for (i = 0; i < NUM; i++) {
        printf("endereco da pessoa %d:\n", i);
        imprime_endereco(residencias[i]);
    }
}
```

Neste programa, o array `residencias` é passado para `obtem_endereco()`, juntamente com o indice onde deve ser guardado o novo endereço. Depois, cada elemento do array é passado para `imprime_endereco()` um por vez.

Observe-se ainda na função `obtem_endereco()` como os membros de cada elemento do array podem ser acessados. elements da estrutura em can be accessed as well. Por exemplo, para acessar a rua do elemento `residencias[0]` usa-se:

```
printf("%s\n", residencias[0].rua);
printf("%s\n", residencias[0].cidade_estado_cep);
```

19.8 Estruturas aninhadas

Como definido anteriormente, membros de estruturas podem ser de qualquer tipo. Isto inclui outras estruturas. Abaixo define-se duas estruturas, a segunda tendo membros que são também estruturas:

```
#define LEN 50

struct endereco {
    char rua[LEN];
    char cidade_estado_cep[LEN];
};

struct student {
    char id[10];
    int idade;
    struct endereco casa;
    struct endereco escola;
};

struct student pessoa;
```

Dadas estas definições, pode-se potencialmente acessar os seguintes campos de `pessoa`, uma variável do tipo `struct student`:

```
pessoa.id
pessoa.casa.rua
pessoa.casa.cidade_estado_cep
pessoa.escola.rua
pessoa.escola.cidade_estado_cep
```

Note o uso repetido de `.` quando se acessa membros dentro de membros.

20 Funções

20.0.1 Funções: o que são e por que usá-las

Quando queremos resolver um problema, em geral tentamos dividi-lo em subproblemas mais simples e relativamente independentes, e resolvemos os problemas mais simples um a um. A linguagem C dispõe de construções (abstrações) que auxiliam o projeto de programas de maneira *top-down*. Uma função cria uma maneira conveniente de encapsular alguns detalhes de "processamento", ou seja, como algum resultado é obtido. Quando esta "computação" é necessária, a função é *chamada*, ou *invocada*. Desta forma, quando uma função é chamada o usuário não precisa se preocupar como a computação é realizada. É importante saber o que a função faz (qual o resultado da execução de uma função) e também como se usa a função. Criando funções, um programa C pode ser estruturado em partes relativamente independentes que correspondem as subdivisões do problema.

Você já viu algumas funções: `printf(), scanf(), getchar(), sqrt()`. Elas são funções de uma biblioteca padrão (do C). Você não sabe como elas foram escritas, mas já viu como utilizá-las. Ou seja, você sabe o *nome* das funções e quais informações específicas você deve fornecer a elas (valores que devem ser *passados* para as funções) para que a função produza os resultados esperados.

Quando nos referirmos a uma função neste texto usaremos a maneira frequentemente utilizada que é o nome da função seguido de `()`.

Tomemos como exemplo o programa abaixo, que calcula o produto de 2 números inteiros positivos apenas se ambos forem primos:

```
/*-------------------------------------------------------------------------
  Verifica se 2 Numeros sao primos e multiplica um pelo outro se o forem
  -------------------------------------------------------------------------
 */

#include <stdio.h>

main()
{

  int n1, n2, j;
  int prod = 0;

  printf("\nEntre com 2 numeros inteiros: ");
  scanf ("%d %d", &n1, &n2);


  /* Se for indicado 1, 0 ou negativo, nao sao primos */
  if ( n1 >= 2 && n2 >= 2) {
    /* Testa se n1 é primo */
    for(j = n1 - 1; (j > 1) && (n1 % j != 0); j--);
    if( j == 1) {        /* n1 eh primo */
      /* Testa se n2 é primo */
      for(j = n2 - 1; (j > 1) && (n2 % j != 0); j--);
      if( j == 1)        /* n2 eh primo */
        prod = n1 * n2;
    }
  }
```

```
  if (prod)
    printf("PRODUTO = %d\n", prod);

}
```

Observe que o código que verifica se um número é primo teve que ser reproduzido dentro do programa por duas vezes (para testar se os números fornecidos pelo usuário eram primos).

Um dos benefícios mais óbvios de usar funções é que podemos evitar *repetição de código*. Em outras palavras, se você quiser executar uma operação mais de uma vez, você pode simplesmente escrever a função uma vez e utilizá-la diversas vezes ao invés de escrever o mesmo código várias vezes. Outro benefício é que se você desejar alterar ou corrigir alguma coisa mais tarde, é mais fácil alterar em um único lugar.

O exemplo acima poderia ser simplificado pela criação de uma função chamada `ehPrimo`, que dado um número n, dá como resultado 1 se este número é primo, ou 0 (zero) se o número não é primo:

```
int ehPrimo(unsigned int n)
{
  int j;

  for(j = n - 1; (j > 1) && (n % j != 0); j--);

  if (j == 1)
    return 1;
  else
    return 0;
}
```

O exemplo pode ser então alterado e simplificado com o uso da função `ehPrimo()`:

```
/*-------------------------------------------------------------------------
  Verifica se 2 Numeros sao primos e multiplica um pelo outro se o forem
  -------------------------------------------------------------------------
 */

#include <stdio.h>

int ehPrimo(unsigned int n)
{
  int j;
  int primo = 0;

  for(j = n - 1; (j > 1) && (n % j != 0); j--);

  if (j == 1)
    primo = 1;

  return primo;
}
```

```
main()
{

  int n1, n2, j;
  int prod = 0, ep1, ep2;

  printf("\nEntre com 2 numeros inteiros: ");
  scanf ("%d %d", &n1, &n2);


  /* Se for indicado 1, 0 ou negativo, nao sao primos */
  if ( n1 >= 2 && n2 >= 2) {
    ep1 = ehPrimo(n1);  /* Verifica se n1 eh primo */
    ep2 = ehPrimo(n2);  /* Verifica se n2 eh primo */

    if (ep1 != 0 && ep2 != 0)
      prod = n1 * n2;
  }

  if (prod)
    printf("PRODUTO = %d\n", prod);

}
```

Como pode ser observado, sejam quais forem os 2 números fornecidos, não precisa escrever um código similar ao mostrado na função `ehPrimo` acima para cada número.Basta `chamar` a função `ehPrimo()`, passar os valores necessários para verificar a primalidade de cada número, e utilizar os resultados.

Evitar repetição de código é a razão histórica que funções foram inventadas (também chamado de procedimento ou subrotinas em outras linguagens de programação). A maior motivação para utilizar funções nas linguagens contemporâneas é a redução da complexidade do programa e melhoria da modularidade do programa. Dividindo o programa em funções, é muito mais fácil projetar, entender e modificar um programa. Por exemplo, obter a entrada do programa, realizar as computações necessárias e apresentar o resultado ao usuário pode ser implementado como diferentes funções chamadas por `main()` nesta ordem.

Funções podem ser escritas *independentemente* uma da outra. Isto significa que, em geral, variáveis usadas dentro de funções não são compartilhadas pelas outras funções. Assim sendo, o comportamento da função é previsível. Se não for assim, duas funções completamente não relacionadas podem alterar os dados uma da outra. Se as variáveis são *locais* a uma função, programas grandes passam a ser mais fáceis de serem escritos. A comunicação entre funções passa a ser controlada – elas se comunicam somente através pelos valores passados as funções e os valores retornados.

20.1 Definindo funções

Um programa C consiste de uma ou mais definições de funções (e variáveis). Há sempre uma função chamada `main`. Outras funções também podem ser definidas. Cada uma pode ser definida separadamente, mas nenhuma função pode ser definida dentro de outra função. Abaixo, mostramos um exemplo simples de um programa que consiste de duas funções: `main()` e `alo()`. Quando executado, este programa imprimirá a mensage `Alo!` três vezes.

```
#include <stdio.h>

/* declaracao do prototipo da funcao alo() */
```

```
void alo(void);

/* definicao da funcao main() */
main()
{
  int i;

  for (i = 1; i<= 3; i++)
    {
      alo();
    }
}

/* definicao da funcao alo() */
void alo(void)
{
   printf("Alo!\n");
}
```

Todas as funções devem ser declaradas antes de serem usadas. As funções da biblioteca padrão, tais como `printf()`, `scanf()` e `getchar()`, são pré-definidas, mas memsmo assim devem ser declaradas (deve ser anunciado ao compilador que elas existem). É por isso que incluímos a linha `#include <stdio.h>` no início do arquivo.

O formato geral da definição de uma função é

tipo-do-resultado nome-da função (*lista-de-argumentos*)
{
        declarações e sentenças
}

A primeira linha da definição é o cabeçalho da função. Ela têm três partes principais: o nome da função, o tipo do resultado (que é um valor) que a função computa e retorna, e entre parênteses uma lista de *parâmetros* (também chamado de *argumentos formais*). Se a função não retorna nenhum valor, o tipo é chamado de `void`, e esta palavra é escrita no cabeçalho na frente do nome da função. Se a função não tiver argumentos formais, a palavra `void` é escrita no lugar da lista de argumentos formais entre os parênteses. Para simplificar a exposição, falaremos sobre o *tipo do retorno* e os *argumentos formais* mais tarde. Eles servem para permitir que as funções troquem informações entre si.

20.2 Funções simples

Para começar, vamos utilizar funções na seguinte forma:

`void` *nome-da-função* (`void`)
{
        declarações e senteças (corpo da função)
}

O primeiro `void` significa que esta função não tem tipo de retorno (não retorna um valor), e o segundo significa que a função não tem argumentos (ela não precisa de nenhuma informação externa para ser exe-

cutada). Isso não significa que a função não faz nada. Ela pode realizar alguma ação, como imprimir uma mensagem. O exemplo abaixo mostra um programa que usa uma função como essa:

```
void alo(void);

main()
{
    alo();
}

void alo(void)
{
    printf("Alo.\n");
}
```

Neste exemplo, o programa consiste de duas funções, `main()` e `alo()`. A ordem em que as funções são definidas não é importante, desde que *protótipos de funções* são usadas. A linha

```
void alo(void);
```

no topo do programa é um *protótipo de função* para a função `alo()`. Um protótipo é usado para declarar uma função. Um protótipo passa ao compilador informações sobre uma função que é definida dentro de um programa "em algum lugar". Protótipos são sempre colocados próximo ao início do programa, antes do começo da definição de funções.

A função `alo()` imprime a mensagem `Alo.` quando chamada. A sentença `printf()` é o corpo da função. Dentro da função `main()` há uma *chamada* a função `alo()`. A função é chamada pelo seu nome seguido de `()` (já que a função `alo` não tem argumentos, nenhuma expressão é escrita dentro dos parênteses). A função `alo()` não retorna um valor, ela é chamada simplesmente para realizar uma ação (imprimir a mensagem). A chamada de função é uma sentença válida em C, portanto deve ser terminada por ponto e vírgula (`;`).

```
alo();
```

Outra coisa que você deve ter notado é que `main()` também é uma função. A função `main()` não difere em nada das demais funções, com a exceção de que contém o programa principal. Além disso, não é necessário declarar o protótipo da função `main()`.

20.2.1 Argumentos

Nosso próximo exemplo pede que o usuário digite suas iniciais, e então chama a função `cumprimenta()` para imprimir a mensagem "Ola" junto com as iniciais digitadas. Estas iniciais (seus valores) são passadas para a função `cumprimenta()`. A função `cumprimenta()` é definida de forma que ela imprimirá a mensagem incluindo quaisquer iniciais passadas.

```
#include <stdio.h>

void cumprimenta(char, char);

main()
{
  char primeiro, segundo;

  printf("Entre com duas iniciais (sem separacao): ");
  primeiro = getchar();
```

```
  segundo = getchar();
  cumprimenta(primeiro, segundo);
}

void cumprimenta(char inic1, char inic2)
{
    printf("Ola, %c%c!\n",inic1,inic2);
}
```

A função `main()` chama a função `cumprimenta()`; `main()` passa para `cumprimenta()` os valores dos dois caracteres para serem impressos. Veja um exemplo de execução do programa:

```
Entre com duas iniciais (sem separacao): YK
Alo, YK!
```

Note que há uma correspondência entre o número e tipo dos valores que `main()` passa (estes são chamados de *parâmetros reais* ou *argumentos reais*) e os argumentos listados no cabeçalho da função `cumprimenta()`.

20.3 Funções que retornam um valor

Funções que não retornam nenhum valor (como `alo()`, `main()`) possuem tipo `void`.

Além de executarem ações (como imprimir) uma função também pode *retornar* um valor para o programa que o chamou. Uma função que retorna um valor tem no cabeçalho o nome do tipo do resultado. O valor retornado pode ser de qualquer tipo, incluindo `int`, `float` e `char` (é claro que uma vez definida, a função só é de um tipo específico). Uma função que retorna um tipo diferente de `void` executa alguns cálculos, e retorna o resultado (que é um único valor) para quem a chamou. A função chamadora pode então usar o resultado. Para retornar um valor para a função chamadora, a função usa a sentença `return`.

O formato da sentença `return` é a seguinte:

`return` *expressão*;

A *expressão* é avaliada e o seu valor é convertido ao tipo de retorno da função (o tipo da função é dado no cabeçalho da função antes do nome da função).

Considere o seguinte exemplo. O programa consiste de duas funções: `main()` e `quadrado`. O programa pede que o usuário digite três números e verifica se eles podem ser os lados de um triângulo reto.

```
/* programa que verifica se 3 numeros podem ser os lados de um
 * triangulo reto.
 */
#include <stdio.h>

int quadrado(int);

main()
{
    int s1, s2, s3;

    printf("Entre tres inteiros: ");
    scanf("%d %d %d", &s1, &s2, &s3);

    if ( s1 > 0 && s2 > 0 && s3 > 0 &&
           (quadrado(s1) + quadrado(s2) == quadrado(s3) ||
```

```
            quadrado(s2) + quadrado(s3) == quadrado(s1) ||
            quadrado(s3) + quadrado(s1) == quadrado(s2)) )
      printf(" %d %d %d podem formar um triangulo reto\n", s1, s2, s3);
   else
      printf(" %d %d %d nao podem formar um triangulo reto\n",s1, s2, s3);
}

/* funcao que calcula o quadrado de um numero */
int quadrado(int n)
   {
      return n * n;
   }
```

Note que quando chamamos a função `quadrado()` passamos o valor no qual desejamos executar o cálculo, e também usamos o valor retornado pela função em expressões. O valor de `quadrado(s1)` é o valor que a função `quadrado()` retorna quando chamado com o valor do argumento sendo igual ao valor da variável `s1`.

Os valores retornados pelas chamadas de funções podem ser usados em todos os lugares valores podem ser usados. Por exemplo,

```
y = quadrado(3);
```

Aqui `quadrado(3)` tem o valor `9`, portanto `9` pode ser atribuído a variável `y`;

```
x = quadrado(3) + quadrado(4);
```

atribuirá `25` a variável `x`, e

```
area = quadrado(tamanho);
```

atribuirá a variável `area` o valor da variável `tamanho` elevado ao quadrado.

O próximo exemplo tem uma função chamada `cinco`:

```
int cinco(void);

main()
{
    printf("cinco = %d\n", cinco() );
}

int cinco(void)
{
    return 5;
}
```

A saída do programa será

```
cinco = 5
```

porque o valor de `cinco()` dentro da sentença `printf()` é `5`. Olhando na sentença `return`, `5` é a expressão retornada para o chamador.

Outro exemplo:

```
int obtem_valor(void);

main()
{
   int a, b;
```

```
    a = obtem_valor();
    b = obtem_valor();

    printf("soma = %d\n", a + b);
}

int obtem_valor(void)
{
    int valor;

    printf("Entre um valor: ");
    scanf("%d", &valor);

    return valor;
}
```

Este programa obtém dois inteiros do usuário e mostra a sua soma. Ele usa a função `obtem_valor()` que mostra uma mensagem e obtém o valor do usuário.

Um exemplo de saída deste programa é:

```
Entre um valor: 15
Entre um valor: 4
soma = 19
```

20.4 Mais sobre o `return`

Quando uma função `return` é executada, a função imediatamente acaba – mesmo que haja código na função após a sentença `return`. A execução do programa continua após o ponto no qual a chamada de função foi feita. Sentenças `return` podem ocorrer em qualquer lugar na função – não somente no final. Também é válido ter mais de um `return` dentro de uma função. A única limitação é que `return` retorna um único valor.

O seguinte exemplo mostra uma função (uma versão para `int` da função `obtem_valor`) que pede para usuário um valor e se o usuário digitar um valor negativo, imprime uma mensagem e retorna um valor positivo.

```
int obtem_valor_positivo(void)
{
    int valor;

    printf("Entre um valor: ");
    scanf("%d", &valor);

    if (valor >= 0)
        return valor;

    printf("Tornando o valor positivo...\n");

    return -valor;
}
```

Em uma função `void`, `return;` (só com `;`) pode ser usado para sair de uma função. O exemplo seguinte, pede instruções ao usuário. Se o usuário reponder **nao**, a função termina. Do contrário, ele imprime as instruções e depois termina.

```
void instrucoes(void)
{
   int ch;

   printf("Voce quer instrucos? (s/n): ");
   ch = getchar();

   /* Termina se resposta for n */
   if (ch == 'n' || ch == 'N')
      return;

   /* Mostra instrucoes */
   printf("As regras do jogo sao . . . ");
     .
     .
     .
   return;
}
```

O `return` final (antes de fechar as chaves do corpo da função) na função é opcional. Se omitido, a função atingirá o final da função e retornará automaticamente. Note que o `return` é opcional somente para funções `void`.

20.5 Mais sobre Argumentos

A comunicação entre uma função e o chamador pode ser nas duas direções. Argumentos podem ser usados pelo chamador para passar dados para a função. A lista de argumentos é definida pelo cabeçalho da função entre parênteses.. Para cada argumento você precisa especificar o tipo do argumento e o nome do argumento. Se houver mais de um argumento, eles são separados por vírgula. Funções que não possuem argumentos tem `void` como lista de argumento. No corpo da função os argumentos (também chamados de *argumentos formais* ou *parâmetros formais*) são tratados como variáveis. É erro defini-los dentro do corpo da função porque eles já estão definidos no cabeçalho. Antes da execução da função os valores passados pelo chamador são atribuídos aos argumentos da função.

Considere o seguinte programa com a função `abs()` que calcula o valor absoluto de um número.

```
int abs(int);

main()
{
    int n;

    printf("Entre um numero: ");
    scanf("%d", &n);

    printf("Valor absoluto de  %d e' %d", n, abs(n));
}

/* Definicao da funcao abs */
int abs(int x)
{
   if (x < 0)
      x = -x;
```

```
    return x;
}
```

A função `abs()` tem um argumento do tipo `int`, e seu nome é `x`. Dentro da função, `x` é usado como uma variável `x`.

Uma vez que `abs()` tem um único argumento, quando ela é chamada, há sempre um valor dentro do parênteses, como em `abs(n)`. O valor de `n` é passado para a função `abs()`, e antes da execução da função, o valor de `n` é atribuído a `x`.

Aqui está um exemplo de uma função que converte uma temperatura de Farenheit para Celsius:

```
float fahr_para_cels(float f)
{
    return 5.0 / 9.0 * (f - 32.0);
}
```

Como você pode ver, esta função tem somente um argumento do tipo `float`. Um exemplo de chamada desta função poderia ser:

```
fervura = fahr_para_cels(212.0);
```

O resultado da função `fahr_para_cels(212.0)` é atribuído a `fervura`. Portanto, depois da execução desta sentença, o valor de `fervura` (que é do tipo `float`) será `100.0`.

O exemplo seguinte possui mais de um argumento:

```
float area(float largura, float altura)
{
    return largura * altura;
}
```

Esta função possui dois argumentos do tipo `float`. Para chamar uma função com mais de um argumento, os argumentos devem ser separados por vírgula. A ordem em que os argumentos são passados deve ser na mesma em que são definidos. Neste exemplo, o primeiro valor passado será a largura e o segundo a altura. Um exemplo de chamada seria

```
tamanho = area(14.0, 21.5);
```

Depois desta sentença, o valor de `tamanho` (que é do tipo `float`) será `301.0`.

Quando passar os argumentos, é importante ter certeza de passá-los na ordem correta e que eles são do tipo correto. Se isso não for observado, pode ocorrer erro ou aviso de compilação, ou resultados incorretos podem ser gerados.

Uma última observação. Os argumentos que são passados pelo chamador podem ser expressões em geral e não somente constantes e variávies. Quando a função é chamada durante a execução do programa, estas expressões são avaliadas, e o valor resultante passado para a função chamada.

20.6 Chamada por valor

Considere novamente a função `quadrado()`. Se esta função é chamada de `main()` como

```
p = quadrado(x);
```

somente o valor (não o endereço) de `x` é passado para `quadrado`. Por exemplo, se a variável tem valor 5, para a função `quadrado()`, `quadrado(x)` ou `quadrado(5)` são o mesmo. De qualquer forma, `quadrado()` receberá somente o valor `5`. `quadrado()` não sabe se na chamada da função o `5` era uma

constante inteira, o valor de uma variável do tipon `int`, ou alguma expressão como `625/25 - 4 * 5`. Quando `quadrado()` é chamado, não interessa qual a expressão entre parênteses, ela será avaliada e o valor passado para `quadrado()`.

Esta maneira de passar argumentos é chamada de *chamada por valor*. Argumentos em C são passados por valor. Portanto, a função chamada não pode alterar o valor da variável passada pelo chamador como argumento, porque ela não sabe em que endereço de memória o valor da variável está armazenado.

20.7 Variáveis locais

Como você provavelmente já reparou em alguns exemplos, é possível definir variáveis dentro de funções, da mesma forma que temos definido variáveis dentro da função `main()`. A declaração de variáveis é feita no início da função.

Estas variáveis são *restritas* a função dentro da qual elas são definidas. Só esta função pode "enxergar" suas próprias variáveis. Por exemplo:

```
void obtem_int(void);

main()
{
    obtem_int();

    /* **** Isto esta' errado  **** */
    printf("Voce digitou %d\n", x);
}

void obtem_int(void)
{
    int x;

    printf("Entre um valor: ");
    scanf("%d", &x);

    printf("Obrigado!\n");
}
```

A função `main()` usou um nome `x`, mas `x` não é definido dentro de `main`; ele é uma variável local a `get_int()`, não a `main()`. Este programa gera erro de compilação.

Note que é possível ter duas funções que usam variáveis locais com o mesmo nome. Cada uma delas é restrita a função que a define e não há conflito. Analise o seguinte programa (ele está correto):

```
int obtem_novo_int(void);

main()
{
    int x;

    x = obtem_novo_int();

    /* ****Isto nao esta errado !! **** */
    printf("Voce digitou %d\n", x);
}
```

```
int obtem_novo_int(void)
{
    int x;

    printf("Entre um valor: ");
    scanf("%d", &x);

    printf("Obrigado!\n");
    return x;
}
```

A função `obtem_novo_int()` usa uma variável local chamada `x` para armazenar o valor digitado e retorna como resultado o valor de `x`. `main()` usa outra variável local, também chamada de `x` para receber o resultado retornado por `obtem_novo_int()`. Cada função tem sua própria variável `x`.

20.8 Protótipos

Os protótipos servem para dar ao compilador informações sobre as funções. Isso para que você possa chamar funções antes que o compilador tenha a definição (completa) das funções. O protótipo de uma função é idêntico ao cabeçalho da função, mas o nome dos argumentos podem ser omitidos e ele é terminado com uma vírgula. Protótipos *declaram* uma função ao invés de *defini-las*. O formato dos protótipos é:

tipo-de-retorno nome-da-função (*lista-dos-tipos-dos-argumentos*) ;

Definindo protótipos, você não precisa se preocupar com a ordem em que define as funções dentro do programa. A principal vantagem de definir protótipos é que erros de chamada de funções (como chamar uma função com o número incorreto de argumentos, ou com argumentos de tipo errado) são detectados pelo compilador. Sem protótipos, o compilador só saberia que há erro depois de encontrar a definição da função. Em versões antigas do compilador C, programas com tais erros compilariam sem erros, o que tornava a depuração de erros mais difícil.

Abaixo, mostramos duas funções e seus protótipos:

```
float volume(float, float, float);
float dinheiro(int, int, int, int);


float volume(float comprimento, float largura, float altura)
{
    return comprimento * largura * altura;
}

float dinheiro(int c25, int c10, int c5, int c1)
{
     return c25 * 0.25 + c10 * 0.10 +
            c5 * 0.05 + c1 * 0.01;
}
```

20.9 Documentação de funções

Você deve documentar as funções que escreve. Na documentação você deve especificar as seguintes informações:

Ação – o que a função faz

Entrada – descrição dos argumentos passados para a função

Saída – descrição do valor retornado pela função

Suposições – o que você assume ser verdade para que a função funcione apropriadamente

Algoritmo – como o problema é resolvido (método)

Estas informações devem ser colocadas como comentário antes da definição da função.

20.10 Comentários

Você pode colocar comentários no seu programa para documentar o que está fazendo. O compilador ignora completamente o que quer esteja dentro de um comentário.

Comentários em C começam com um `/*` e terminam com um `*/`. Alguns exemplos:

```
/* Este e' um comentario sem graca */

/* Este e'
   um comentario
   que usa
   diversas linhas
 */

/* Este e'
 * um comentario
 * de diversas linhas
 * mais bonito
 */
```

Note que não podemos aninhar comentários dentro de comentários. Um comentário termina no primeiro `*/` que encontrar. O comentário abaixo é ilegal:

```
/* Este e' um comentario /* illegal */ ilegal */
```

Regras para comentário

É sempre uma boa idéia colocar comentários em seu programa das coisas que não são claras. Isto vai ajudar quando mais tarde você olhar o programa que escreveu já há algum tempo ou vai ajudar a entender programas escritos por outra pessoa.

Um exemplo de comentário útil:

```
/* converte temperatura de farenheit para celsius */
celsius = (fahrenheit - 32) * 5.0 / 9.0;
```

O comentário deve ser escrito em português e não em C. No exemplo abaixo

```
/* usando scanf, obter valor de idade e multiplicar por 365 para
 * obter dias */
```

```
  scanf("%d", &idade);
  dias = idade * 365;
```

o comentário é basicamente uma transcrição do código do programa. Em seu lugar, um comentário como

```
  /* obtem idade e transforma em numero de dias */
```

seria mais informativo neste ponto. Ou seja, você deve comentar o código, e não codificar o comentário.

Você também deve evitar comentários inúteis. Por exemplo:

```
  /* Incrementa i */
  i++;
```

Não há necessidade de comentários já que `i++` já é auto explicativo.

E abaixo está um exemplo de como você deve comentar uma função.

```
/* funcao instrucoes()
 *  acao:        mostra instrucoes do programa
 *  entrada:     nenhuma
 *  saida:       nenhuma
 *  suposicoes:  nenhuma
 *  algoritmo:   imprime as instrucoes
 */
void instrucoes(void)
{
   /* mostra instrucoes */
   printf("O processo de purificacao do  Uranio-235 e' . . .  ");
     .
     .
}
```

21 O pré-processador

O pré-processador é um programa que faz alguns processamentos simples antes do compilador. Ele é executado automaticamente todas as vezes que seu programa é compilado, e os comandos a serem executados são dados através de *diretivas do pré-processador*.

Estas diretivas são colocadas em linhas que contém somente a diretiva (elas não são código da linguagem C, portanto as regras para elas são um pouco diferentes). As linhas que começam com um `#` são comandos para o pré-processador. A linha inteira é reservada para este comando (nenhum código C pode aparecer nesta linha e comandos do pré-processador não podem estar separados em diversas linhas).

21.1 A diretiva `#define`

Uma diretiva que é usada frequentemente é o `#define`. Esta diretiva é usada para fazer *substituição de macros*. Por enquanto, mostraremos uma utilização simples do `#define`, que é simplestemente uma substituição no texto.

O uso mais frequente desta diretiva é dar nomes simbólicos a uma constante (você já viu outra maneira de definir contantes que é colocar a palvavra `const` antes da definição de uma variável). Por exemplo, seria conveniente usar `PI` em seus programas ao invés de digitar `3.1415926535` toda hora. Como outro exemplo, se você quiser escrever um programa sobre estudantes de uma turma de 81 alunos, você poderia definir `NUM_ALUNOS` como `81`. Assim, se o número de alunos mudar, você não precisaria modificar todo o seu programa onde o número de alunos (`81`) é utilizado, mas simplesmente alterar a diretiva `#define`. Estas duas diretivas são definidas da seguinte forma:

```
#define PI          3.1415926535
#define NUM_ALUNOS  81
```

Por convenção, nomes introduzidos por um `#define` são geralmente em letra maiúscula (e variáveis são em letra minúscula, ou uma mistura de letras minúsculas e maiúsculas). Assim, quando você vê um nome em um programa, você sabe se o nome refere-se a uma variável ou um nome definido por um `#define`.

Considere o seguinte programa exemplo que usa `PI`:

```
#define PI        3.14159265

main()
{
   double raio;

   printf("Entre com o raio: ");
   scanf("%f", &raio);

   printf("Circunferencia = %f\n", 2.0 * PI * raio);
}
```

Lembre-se que o nome `PI` não é um nome de variável. Ele é um nome que o pré-processador substituirá pelo *texto* especificado pelo `#define` (mais ou menos da mesma forma que o comando pesquisa-e-substitui do editor de texto). O compilador nunca vê ou sabe sobre `PI`. O compilador vê o seguinte `printf()` do programa acima depois do pré-processador ser executado:

```
   printf("Circunferencia = %f\n", 2.0 * 3.14159265 * raio);
```

21.2 A diretiva `#include`

Agora imagine que estamos escrevendo uma biblioteca geométrica: um conjunto de funções para calcular a área de cilindros, cones, esferas. Se diferentes pessoal estão escrevendo cada uma das funções, eles

provavelmente colocarão suas funções em diferentes arquivos. Mas todas as funções usam o numero π, e algumas outras constantes podem ser necessárias também. Ao invés de colocar o `#define` no início de cada arquivo, um único arquivo `geom.h` pode ser criado. Este arquivo conterá a linha

```
#define  PI    3.14159265
```

Assim, se todos os arquivos de funções geométricas puderem enxergar `geom.h`, eles compartilharão as mesmas definições. é para isso que usamos a diretiva `#include`, para incluir em seu programa, informações que estão em outro arquivo. Estas diretivas geralmente estão no início do programa fonte, antes da definição de funções e varáveis. Por exemplo, a diretiva

```
#include "geom.h"
```

colocada nos arquivos fontes que contêm as funções geométricas fará com que todos eles usem o nome simbólico `PI` ao invés de `3.14159265`. O fato do nome do arquivo estar em aspas significa que o arquivo `geom.h` está no mesmo diretório que os arquivos fontes (ao invés do diretório onde se encontram as bibliotecas padrão de C).

A diretiva

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

é colocada no início do programa fonte para incluir informações (como protótipos de funções) que são necessários quando `printf()` e `scanf()` são chamados dentro do programa. O arquivo entre `< >` está em algum diretório padrão conhecido pelo pré-processador. Este arquivo `stdio.h` é comum a todas as implementações da linguagem C e contém infomações necessárias para executar operações de entrada e saída da entrada e saída padrão (teclado e monitor).

A extensão `.h` vem do inglês *header file*. Apesar de não ser obrigatório que arquivos incluídos tenham a extensão `.h`, geralmente esta é a convenção utilizada.

21.3 Comentários

De um modo geral, o pré-processador dos compiladores existentes remove todos os comentários do arquivo fonte antes do programa ser compilado. Portanto, o compilador nunca vê realmente os comentários.

22 Mais sobre funções

A ênfase aqui será em como funções funcionam. O que acontece quando uma função é chamada ? A que variável um nome está se referenciando?

O tratamento em tempo de execução de um nome de variável em C é simples: um nome de variável é ou uma variável local (a função) ou uma variável global (definida fora de qualquer função).

Em C, todas as funções tem que ser definidas. Para cada função deve ser definido um protótipo. O protótipo é escrito fora de qualquer função. Desta forma, nomes de funções são visíveis para todas as outras funções que podem então invocá-las. A função `main()` é especial: é onde a execução do programa começa, e o protótipo de `main()` pode ser omitido.

Uma definição de função consiste de quatro partes:

1. o nome da função;
2. a lista de parâmetros formais (argumentos) com seus nomes e tipos. Se não houver argumentos, a palavra `void` é escrita entre os parênteses.
3. o tipo do resultado que a função retorna através da sentença `return` ou `void` se a função não retorna nenhum valor. Lembre-se que somente um valor pode ser retornado por uma sentença `return`.
4. o corpo da função, que é uma sentença composta (começa e termina com chaves (`{ }`) contendo definição de variáveis e outras sentenças. Em C, não se pode definir uma função dentro de outra.

Para funções com argumentos: uma função é chamada dando o seu nome e uma lista de argumentos (expressões que são avaliadas e cujos valores são atribuídos para os correspondentes parâmetros formais da função).

Por exemplo, suponha que `triang_area()` e `circ_area()` sejam funções que calculam a área de triângulos e círculos, respectivamente. Seus protótipos são:

```
float triang_area(float , float);
float circ_area(float);
```

Estas funções podem chamadas de dentro de outras funções. Os argumentos reais com os quais elas são chamadas podem ser expressões constantes, or variáveis locais, ou qualquer expressão cuja avaliação resulte em valores do tipo `float` (inteiros são convertidos para `float` da mesma forma que ocorre com atribuição de inteiros para variáveis do tipo `float`). Alguns exemplos de chamadas:

```
float  area2, area3, area4, area5, base, altura, raio;

printf("area do triangulo = ", triang_area(0.03, 1.25));
base = 0.03;
altura = 1.25;
area2 = triang_area(base, altura);
area3 = triang_area(1.6, altura);
area4 = triang_area( 0.03 + base, 2 * altura);
raio = base + altura;
area5 = triang_area(raio, circ_area(raio));
```

A última sentença do exemplo acima atribui a variável `area5` a área de um triângulo cuja base é igual ao valor da variável `raio` e a altura é igual a area de um círculo de raio igual ao valor da variável `raio`.

Quando um programa é executado, somente uma única função tem o controle em determinado momento. Falaremos mais sobre o que acontece quando uma função é chamada mais tarde nestas notas de aula.

Variáveis Locais **Variáveis que são definidas dentro de uma função são variáveis locais desta função. Parâmetros formais de uma função são variáveis locais da função.** Variáveis locais são privativas a função na qual são definidas. Somente esta função pode enxergá-las (ela conhece o endereço das variáveis e pode usar e modificar o seu conteúdo). Nenhuma outra função pode acessar variáveis locais de outra função sem permissão (uma função pode acessar variáveis locais de outra se esta passar o endereço da variável local como argumento – este assunto será tratado em notas de aula futuras). O fato de cada função manter variáveis locais "escondidas" do resto do mundo torna mais fácil a tarefa de escrever programas estruturados e modulares. Quando você está escrevendo uma função, você pode dar as suas variáveis locais o nome que quiser. Você também não precisa se preocupar se outra pessoa escrevendo outra função terá acesso ou altera variáveis locais a sua função.

Variáveis locais que são definidas dentro da função devem ser inicializadas com algum valor antes de serem usadas. Caso contrário, o seu valor é indefinido.

Já que parâmetros formais (argumentos) são variáveis locais da função, eles podem ser usados no corpo da função. Eles não devem ser definidos dentro da função (sua definição já está no cabeçalho da função). Os parâmetros formais não precisam ser inicializados. Seus valores são fornecidos pelo chamador da função através dos argumentos reais.

Considere o seguinte exemplo:

```
/*************************************************************
 * Um programa que calcula a area de triangulos e circulos.
 * A base, altura e raio sao fornecidos pelo usuario.
```

```
 * A saida do programa e a area do triangulo e circulo.
 ************************************************************/

#include <stdio.h>

#define PI 3.1415

/*******************
    prototipos
*******************/
float triang_area(float, float);
float circ_area(float);

/*******************
definicao de funcoes
*******************/

main(void)
{
  /* definicao das variaveis locais */
  float   base, altura, raio;

  /* dialogo de entrada */
  printf("\nEntre com a base e altura do triangulo: ");
  scanf("%f %f", &base, &altura);
  printf("\nEntre com o raio do circulo: ");
  scanf("%f", &raio);

  /* chama as funcoes e imprime o resultado */
  printf("Area do triagulo com base e altura %f e %f = %f\n",
         base, altura, triang_area(base, altura));
  printf("Area do circulo com raio %f = %f\n", raio, circ_area(raio));
}

/************************************************************
 * funcao: triang_area
 * calcula a area de um triangulo dada a base e altura
 * Entrada: base e altura do triangulo
 * Saida: area do triangulo
 ************************************************************/
float triang_area(float base, float alt)
{
  return 0.5*base*alt;
}

/************************************************************
 * funcao: circ_area
 * calcula a area de um circulo dado o raio
 * Entrada: raio do circulo
 * Saida: area do circulo
```

```
 ************************************************************/
float circ_area(float r)
{
  return PI*r*r;
}
```

Este programa C consiste de três funções, `main()`, `triang_area()`, e `circ_area()`. `main()` tem variáveis locais chamadas `base, altura` e `raio`; `triang_area()` tem como variáveis locai seus parâmetros formais, `base` e `alt`; `circ_area()` tem como variável local seu parâmetro formal `r`.

Em geral, uma variável local só existe durante a execução da função na qual ela está definida. Portanto, variáveis locais existem desde o momento que a função é chamada até o momento em que a função é completada. Tais variáveis são chamadas de `automatic`. Em C, uma variável pode ser definida como sendo `static`. Neste caso, uma variável local não é visível de fora do corpo da função, mas ela não é destruída no final da função como variáveis automáticas são. Cada vez que a função é chamada, o valor das variáveis *static* é o valor final da variável da chamada anterior.

Variáveis Globais

Até este momento, todas as variáveis que vimos são definidas dentro de funções (no corpo da função ou como parâmetros formais). é possível também definir variáveis *fora* das funções. Tais variáveis são chamadas de *variáveis globais* ou *externas*. O formato da definição de variáveis globais é o mesmo da definição de variáveis locais. A única diferença é *onde* a variável é definida: variáveis globais são definidas fora de qualquer função. Ao contrário das variáveis locais, variáveis globais podem ser vistas por todas as funções definidas após a definição das variáveis globais.

Nós temos usado declarações "globais" este tempo todo – por exemplo, as declarações de protótipos de funções. Elas são declaradas fora de qualquer função e podem ser vistas por qualquer função que estão após sua declaração.

No exemplo seguinte, uma variável `saldo` que é atualizada por três funções diferentes é definida como uma variável global. As três funções que a atualizam não chamam uma a outra.

```
/*************************************************************
 * Caixa eletronico simples
 * o saldo e o valor a ser alterado e entrado pelo usuario
 * a saida do programa e' o saldo atualizado, incluindo juros
 *************************************************************/

#include <stdio.h>

#define JUROS 0.07

/*******************
    prototipos
*******************/
void credito(float);
void debito(float);
void juros(void);

/*******************
      globais
 *******************/
float saldo;  /* saldo atual;
```

```
                * Alterada em: credito(), debito(), juros(), main()
                * Lida em:
                */


/***********************
  definicao de funcoes
 ***********************/

main(void)
{
   float valor;                         /* valor a ser  depositado/retirado  */

   printf("Entre com o saldo atual: ");
   scanf("%f",&saldo);
   printf("Deposito: ");
   scanf("%f", &valor);
   credito(valor);
   printf("Retirada: ");
   scanf("%f", &valor);
   debito(valor);
   juros();
   printf("Juros 7%%.\n");
   printf("Saldo = :  %.2f\n ", saldo);
}

/*****************************************************************
 * Deposita um valor; atualiza a variavel global saldo
 * Entrada: valor a ser depositado
 * Saida: nenhum
 *****************************************************************/
void credito(float val)
{
   saldo += val;
}

/*****************************************************************
 * Debita um valor; atualiza a variavel global saldo
 * Entrada: valor a ser debitado
 * Saida: nenhum
 *****************************************************************/
void debito(float val)
{
   saldo -= val;
}

/*****************************************************************
 * Acumula juros; atualiza a variavel global saldo; juros: RATE
 * Entrada: nenhuma
 * Saida: nenhuma
```

```
 ***************************************************************/
void juros(void)
{
  saldo += (saldo * JUROS);
}
```

Um exemplo de execução do programa:

```
Entre com o saldo atual: 1000
Deposito: 200
Retirada: 80
Juros 7%.
Saldo =  1198.40
```

Variáveis globais devem ser usadas **SOMENTE** quando muitas funções usam muito as mesmas variáveis. No entanto, o uso de variáveis globais é perigoso (e não recomendado) porque a modularidade do programa pode ser afetada. Uma variável global pode ser alterada de dentro de uma função, e esta alteração pode influir no resultado de uma outra função, tornando-a incorreta (em um exemplo dado posteriormente nestas notas, duas chamadas a função `soma_y()` com o mesmo argumento (zero) produz resultados diferentes, `100` e `300`).

Quando variáveis globais são utilizadas, deve ser dado a elas nomes descritivos e um breve comentário qual a finalidade da variável e quais funções a acessam.

Neste curso, você utilizará variáveis globais SOMENTE QUANDO FOR DADO PERMISSãO PARA FAZê-LO. Caso contrário, não é permitido utilizá-las (ou seja, serão descontados pontos).

Escopo de Variáveis

Como já discutimos anteriormente, uma variável é uma abstração de dados que nós usamos em um programa. A variável representa um endereço de memória onde os valores são armazenados. Durante a execução do programa, valores diferentes poder ser armazenados neste endereço. Quando uma variável é definida, o nome da variável é "atrelada" a um endereço específico na memória. Até este momento, já discutimos o que é o nome de uma variável, seu endereço, tipo e valor. Outra característica que apresentaresmo agora é o `escopo`. O escopo de uma variável refere-se a parte do programa onde podemos utilizar a variável. Em outras, palavras, uma variável é "visível" dentro do seu escopo.

O escopo de uma variável local é a função na qual ela é definida. Os parâmetros formais de uma função também são tratados como variáveis locais.

O escopo de uma variável global é a porção do programa depois da definição da variável global (a partir do ponto onde ela é definida até o final do programa).

Se o nome de uma variável global é idêntico a uma variável local de uma função, então dentro desta função em particular, o nome refere-se a variável local. (Embora tais conflitos devem ser evitados para evitar confusão).

Por exemplo, considere o seguinte programa:

```
int valor = 3;    /* definicao da variavel global */

main()
{
   /* definicao local de  valor */
   int valor = 4;
   printf("%d\n", valor);
}
```

A saída do programa acima será 4 já que valor refere-se a definição local.
Considere outro exemplo:

```
#include <stdio.h>

int soma_y(int);
int soma_yy(int);
int y = 100;      /* variavel global  */

main(void)
{
  int z = 0;      /* variavel local */

  printf("%d\n", soma_y(z));
  printf("%d\n", soma_yy(z));
  printf("%d\n", soma_y(z));
}

int soma_y(int x)
{
  return x + y;   /* x e' variavel local, y e' global */
}

int soma_yy(int x)
{
  y = 300;        /* y e' variavel global */
  return x + y;   /* x e' variavel local  */
}
```

Vamos seguir a execução deste programa. Primeiro, a variável global y é criada e inicializada com 100. Então, a execução da função main() comeca: é alocado espaço na memória para a variável local z. Esta variável é inicializada com 0. Considere a primeira sentença printf():

```
printf("%d\n", soma_y(z));
```

Esta é uma chamada para a função da biblioteca padrão printf(). Os parâmetros reais desta chamada são o string "%d\n" e a expressão soma_y(z). A última expressão é a chamada da função soma_y(). O valor desta expressão é o resultado retornado por soma_y(). Qual o resultado? A função soma_y é chamada com o parâmetro real z. Como z = 0, este é o valor que será passado para a função soma_y; o 0 é copiado para o parâmetro formal x da função soma_y(). Portanto, durante a excução da primeira chamada a função soma_y(), o valor da expressão x + y será 0 + 100, que é 100. Portanto, o valor da primeira chamada soma_ y(z) é 100, e este número será impresso com o primeiro printf() em main(). Agora considere a segunda senteça:

```
printf("%d\n", soma_yy(z));
```

Quando a função soma_yy(z) é chamada, o valor de z (a variável local z) ainda é 0, portanto novamente 0 é copiado para o parâmetro formal int x da função soma_yy. Quando a execu,ão de soma_yy() começa, ela primeiro troca o valor da variável global y para 300 e então retorna o valor de x + y, que neste caso é 0 + 300. Portanto, o valor desta chamada a soma_yy(z) é 300, e este número será impresso pelo segundo printf() em main().
Por último, considere a terceira senteça:

```
printf("%d\n", soma_y(z));
```

Quando a função `soma_y(z)` é chamada, o valor de `z` ainda é `0`, portanto, `0` é copiada para o parâmetro formal `int x` da função `soma_y()`. Quando `soma_ y()` é executada pela segunda vez, a variável global `y` foi modificada para `300`, portanto o valor de `x + y` é `0 + 300`. Portanto, o valor da chamada `soma_yy(z)` é `300`, e este número será impresso pelo terceiro `printf()` em `main()`.

Portanto, a saída da execução deste programa será

```
100
300
300
```

Neste exemplo, o escopo da variável global `y` é o programa todo.

O escopo da variável local `z`, definida dentro de `maio` é o corpo da função `main`. O escopo do parâmetro formal `x` da função `soma_y` é o corpo de `soma_y`. O escopo do parâmetro formal `x` da função `soma_yy` é o corpo de `soma_yy`.

22.1 Outro exemplo

Aqui apresentamos um exemplo de uma função mais complicada. Esta função calcula a "raiz quadrada inteira" de um número (o maior inteiro menor ou igual a raiz quadrada do número).

Este programa usa o algoritmo "divide e calcula média" (uma aplicação do método de Newton). Ele executa o seguinte:

Dado x, achar $\sqrt{x}$ computando sucessivamente

$$a_n = \begin{cases} 1 & \text{se } n = 0 \\ \frac{\frac{x}{a_{n-1}} + a_{n-1}}{2} & \text{caso contrário} \end{cases} \quad \text{para todo } n \,\epsilon\, \mathcal{N}$$

Os valores de a_n convergem para $\sqrt{x}$ a medida que n cresce.

Para achar a raiz quadrada inteira, este algoritmo é repetido até que

$$a_n^2 \leq x < (a_n + 1)^2$$

Por exemplo, para achar a raiz quadrada inteira de 42 (usando divisão inteira que trunca a parte fracional do número)
$a_0 = 1$, $a_1 = (42/1 + 1)/2 = 21$, $a_2 = (42/21 + 21)/2 = 11$, $a_3 = (42/11 + 11)/2 = 7$, $a_4 = (42/7 + 7)/2 = 6$.
Uma vez que $a_4^2 = 6^2 = 36 \leq 42 < (a_4 + 1)^2 = 7^2 = 49$, o processo termina e a resposta é 6.

(Não é necessário você entender por que este algoritmo funciona – portanto não se preocupe se não conseguir entendê-lo)

```
int raizInteira(int);          /* prototipo */

/*************************************************************
 * function: raizInteira(x)
 *  acao:        dado x, retorna a raiz quadrada inteira de x
 *  in:          inteiro positivo x
 *  out:         raiz quadrada inteira de x
 *  suposicoes:  x >= 0
 *  algoritmo:   metodo de dividr e calcular media:  comecando com
 *               um palpite de 1, o proximo palpite e' calculado como
 *               (x/palpite_ant + palpite_ant)/2.  Isso e' repetido
 *               ate' que palpite^2 <= x < (palpite+1)^2
 *************************************************************/
int raizInteira(int x)
{
    int palpite = 1;

    /* Continue ate' que o  palpite esteja correto */
    while (!(x >= palpite*palpite  &&  x < (palpite+1)*(palpite+1))) {
        /* Calcula proximo palpite */
        palpite = (x/palpite + palpite) / 2;
    }
    return palpite;
}
```

Note que usando a lei de DeMorgan, podemos re-escrever a expressão teste do `while` em uma forma equivalente:

```
x < palpite * palpite || x >= (palpite + 1) * (palpite + 1)
```

Deve estar claro neste ponto a diferenca entre ação e algoritmo. Uma pessoa que quer usar esta função precisa saber somente a ação, não o algoritmo. é também importante especificar os dados que são esperados pela função e retornados por ela para outras pessoas poderem usá-la. As suposições devem esclarecer as restrições da função sobre quando a função pode falhar ou produzir resultados errados. Neste caso, um número negativo produziria um erro, já que números negativos não possuem raiz quadrada.

Não há necessidade de ir em muitos detalhes em qualquer parte da documentação da função. Embora ela deva conter informação suficiente para que alguém (que não possa ver o código) saber utilizá-la. Detalhes sobre implementação e detalhes menores sobre o algoritmo devem ser colocados como comentários no próprio código.

23 Ativação de função

Uma função começa sua execução assim que for chamada. Cada execução da função é chamada de ativação da função. Como já mencionamos em notas de aula anteriores, variáveis locais são locais a ativação da função: cada ativação possui suas próprias variáveis locais. No começo da ativação, memória é alocada para as variáveis locais e no final da execução, elas são dealocadas.

Definições de funções em C não podem ser aninhadas, mas ativações de função podem: uma função, digamos A, pode chamar uma outra função, digamos B (dizemos que A chama B). Nos referimos a A como o "chamador" e B como a função "chamada".

O que acontece quando uma função chama outra (quando A chama B)? Um registro especial, chamado *registro de ativação* é criado. A informação neste registro é necessária para a ativação da função chamada e para a reativação do chamador depois que a execução da função chamada termina.

1. C usa chamada-por-valor, ou seja, o chamador avalia as expressões que são os parâmetros reais e passa seus valores para a função chamada.
2. A informação necessária para reiniciar a execução da função chamadora é guardada em um registro de ativação. Tal informação inclui o endereço da instrução do chamador que será executada depois que a função chamada termine.
3. A função chamada aloca espaço na memória para suas variáveis locais.
4. O corpo da função chamada é executado.
5. O valor retornado para a função chamadora através de um `return` é colocado em um lugar especial para que a função chamadora possa encontrá-lo. O controle retorna a função chamadora.

O fluxo de controle através de ativação de funções é da forma *último-que-entra-primeiro-que-sai*. Se A chama B e B chama C: A é ativado primeiro, então B é ativado (um registro de ativação para "A chama B" é criado e armazenado, A é temporariamente suspenso), então C é ativado (um registro de ativação de "B chama C" é criado e armazenado, A e B são suspensos); C é o último a iniciar execução, mas o primeiro a terminar (último-a-entrar-primeiro-a-sair). Depois que C termina, B é reativado. O registro de ativação "B chama C" foi criado por último, mas o primeiro a ser destruído (no momento que o controle é retornada para B). Depois que B termina, A é reativado. O registro de ativação correspondente a "A chama B" é destruído no momento em que o controle retorna para A.

24 Mais sobre funções: Quando `return` não é suficiente

Considere o programa abaixo que pede ao usuário dois inteiros, armazena-os em duas variáveis, troca seus valores, e os imprime.

```
#include <stdio.h>

main(void)
{
  int a, b, temp;

  printf("Entre dois numeros: ");
  scanf("%d %d", &a, &b);

  printf("Voce entrou %d e %d\n", a, b);
```

```
  /* Troca a com b */
  temp = a;
  a = b;
  b = temp;

  printf("Trocados, eles sao %d e %d\n", a, b);
}
```

Aqui está um exemplo de execução do programa:

```
Entre dois numeros: 3 5
Voce entrou 3 e 5
Trocados, eles sao 5 e 3
```

O seguinte trecho do programa executa a troca de valores das variáveis `a` e `b`:

```
temp = a;
a = b;
b = temp;
```

É possível escrever uma função que executa esta operação de troca? Considere a tentativa abaixo de escrever esta função:

```
#include <stdio.h>

void troca(int, int);

void troca(int x, int y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}

main(void)
{
    int a, b;

    printf("Entre dois numeros: ");
    scanf("%d %d", &a, &b);

    printf("Voce entrou com %d e %d\n", a, b);

    /* Troca a e b */
    troca(a, b);

    printf("Trocados, eles sao %d e %d\n", a, b);
}
```

Se você executar este programa, verá que ele **não** funciona:

```
Entre dois numeros: 3 5
Voce entrou 3 e 5
Trocados, eles sao 3 e 5
```

Como você já viu nas notas anteriores, em C os argumentos são passados **por valor**. Uma vez que somente os valores das variáveis são passados, não é possível para a função `troca()` alterar os valores de `a` e `b` porque `troca()` não sabe onde na memória estas variáveis estão armazenadas. Além disso, `troca()` não poderia ser escrito usando a sentença `return` porque só podemos retornar um valor (não dois) através da sentença `return`.

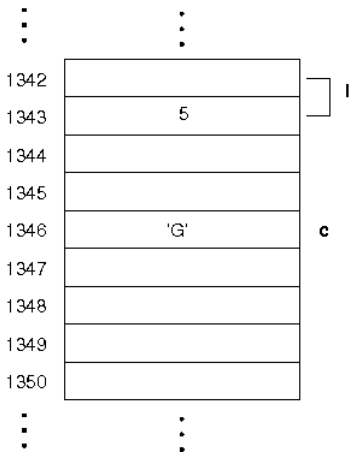
24.1 Usando ponteiros

A solução para o problema acima é ao invés de passar os valores de `a` e `b`, passar o endereço das variáveis `a` e `b`. Desta forma, `troca()` saberia em que endereço de memória escrever, portanto poderia alterar os valores de `a` e `b`.

Lembre-se que em C a cada variável está associado: (i) um nome; (ii) um tipo; (iii) um valor; e (iv) um endereço. Assuma que existam as seguintes definições de variáveis.

```
int i = 5;
char c = 'G';
```

Na memória, eles podem estar armazenados da forma abaixo:



A variável inteira `i` está armazenada no endereço `1342`. Ela usa dois bytes de memória (quando um objeto usa mais de um byte, seu endereço é onde ele começa – neste caso, `1342` e não `1343`). A variável do tipo `char c` está armazenada no endereço `1346` e usa um byte de memória. O compilador é que controla do local de armazenamento destas variáveis em memória.

24.2 O operador de endereço (`&`)

Nós podemos usar o operador de endereço para determinar o endereço de uma objeto na memória. Este operador só pode ser usado com "lvalues" (objetos que podem estar no lado esquerdo de uma atribuição, como no caso de variáveis) porque "lvalues" tem um endereço alocado na memória.

Por exemplo, no exemplo acima, poderíamos usar o operador de endreço como nas expressões abaixo:

`&i` tem valor `1342`
`&c` tem valor `1346`

24.3 Tipo ponteiro

Em C, uma variável que contém um endereço de memória é uma variável do tipo ponteiro. Um valor, que é um endereço (como `&a`) é um valor de ponteiro. Quando um ponteiro (a variável) contém um determinado endreço, dizemos que ele *aponta* para o endereço de memória (ou se este endereço de memória for associado a uma variável, dizemos que ele aponta para esta variável).

Há um tipo distinto de ponteiro para cada tipo básico C (como `int`, `char` e `float`). É verdade que todos os endereços tem o mesmo tamanho (em Turbo C são 2 bytes), mas nós também precisamos saber algo sobre o que é armazenado no endereço de memória apontado (quantos bytes ocupa e como os bytes devem ser interpretados). Por exemplo, um tipo ponteiro usado para "apontar" para inteiros é chamado *ponteiro para int* e isso é denotado por um `*`. Variáveis do tipo ponteiro para int são usados para armazenar endereços de memória que contem valores do tipo `int`.

Por exemplo, dadas as definições de `i` e `c` acima, nós podemos definir duas novas variáveis `pi` e `pc`, ambos do tipo ponteiro.

```
int *pi;
char *pc;
```

Nesta definição não inicializamos as variáveis com nenhum valor. Podemos inicializá-las com:

```
pi = &i;
pc = &c;
```

Depois destas atribuições, o valor de `pi` seria `1342`, e o valor de `pc` seria `1346`.

Note que nesta definição da variável `int *pi`, `pi` é o nome da variável e `int *` é o tipo de `pi` (ponteiro para `int`).

24.4 O operador de dereferência: `*`

Quando um ponteiro aponta para um endereço de memória, a operação para acessar o conteúdo do endereço apontado é chamado de dereferência. O operador unário `*` é usado para fazer a dereferência. Note que este uso do símbolo `*` não tem nada a ver com o símbolo de multiplicação. Usando os exemplos anteriores, `*pi` é o objeto apontado por `pi`.

`*pi` tem valor `5`
`*pc` tem valor `'G'`

Como um pointer dereferenciado (tais como `*pi` ou `*pc`) refere-se a um objeto na memória, ele pode ser usado não só como valor, mas também como um "lvalue". Isto significa que um pointer dereferenciado pode ser usado no lado esquerdo de uma atribuição. Veja alguns exemplos:

```
printf("Valor= %d, Char = %c\n", *pi, *pc);
*pi = *pi + 5;
*pc = 'H';
```

`*pi` no lado esquerdo do = refere-se ao endereço de memória para o qual `pi` aponta. `*pi` no lado direito do = refere-se ao valor armazenado no endereço apontado por `pi`. A sentença `*pi = *pi + 5;` faz com que o valor armazenado no endereço apontado por `pi` seja incrementado de `5`. Note que o valor de `*pi` muda, não o valor de `pi`.

Neste exemplo, os valores das variáveis `i` e `c` poderiam ter sido alterados sem a utilização de ponteiros da seguinte forma:

```
printf("Valor = %d, Char = %c\n", i, c);
i = i + 5;
c = 'H';
```

Os exemplos acima ilustram como uma variável pode ser acessada diretamente (através do seu nome) ou indiretamente (através de um ponteiro apontando para o endereço da variável).

24.5 Ponteiros como argumentos de funções

Nos exemplos acima, pode parecer que ponteiros não são úteis, já que tudo que fizemos pode ser feito sem usar ponteiros. Agora, considere novamente o exemplo da função `troca()`. Quando `a` e `b` são passados como argumentos para `troca()`, na verdade, somente seus valores são passados. A função não podia alterar os valores de `a` e `b` porque ela não conhece os endereços de `a` e `b`. Mas se ponteiros para `a` e `b` forem passados como argumentos ao invés de `a` e `b`, a função `troca()` seria capaz de alterar seus valores; ela saberia então em que endereço de memória escrever. Na verdade, a função não sabe que os endereços de memória são associados com `a` e `b`, mas ela pode modificar o conteúdo destes endereços. Portanto, passando um ponteiro para uma variável (ao invés da variável), habilitamos a função a alterar o conteúdo destas variáveis da função chamadora.

Uma vez que endereços de variáveis são do tipo ponteiro, a lista de parâmetros formais da função deve refletir isso. A definição da função `troca()` deveria ser alterada, e a lista de parâmetros formais deve ter argumentos não do tipo `int`, mas ponteiros para `int`, ou seja, `int *`. Quando chamamos a função `troca()`, nós não passamos como parâmetros reais `a` e `b`, que são do tipo `int`, mas `&a` e `&b`, que são do tipo `int *`. Dentro da função `troca()` deverá haver mudanças também. Uma vez que agora os parâmetros formais são ponteiros, o operador de dereferência, `*`, deve ser usado para acessar os objetos. Assim, a função `troca()` é capaz de alterar os valores de `a` e `b` "remotamente".

O programa abaixo é a versão correta do problema enunciado para a função `troca()`:

```
#include <stdio.h>

void troca(int *, int *);

/* function troca(px, py)
 *  acao:        troca os valores inteiros apontados por px e py
 *  entrada:     apontadores px e py
 *  saida:       valor de  *px e *py trocados
 *  suposicoes:  px e py sao apontadores validos
 *  algoritmo:   primeiro guarda o primeiro valor em um temporario e troca
 */
void troca(int *px, int *py)
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}

main(void)
{
    int a, b;

    printf("Entre dois numeros: ");
    scanf("%d %d", &a, &b);

    printf("Voce entrou com %d e %d\n", a, b);

    /* Troca a e b -- passa enderecos */
    troca(&a, &b);
```

```
    printf("Trocados, eles sao %d e %d\n", a, b);
}
```

A saída deste programa é:

```
Entre dois numeros: 3 5
Voce entrou com  3 e 5
Trocados, eles sao 5 e 3
```

Basicamente, se a função precisa alterar o valor de uma variávels da função chamadora, então pssamos o endereço da variável como parâmetro real, e escrevemos a função de acordo, ou seja, com um ponteiro como parâmetro formal.

24.6 Precedência de operadores

A precedência dos operadores `*` e `&` é alta, a mesma que outros operadores unários. A tabela 5 apresenta a precedência de todos os operadores vistos até agora.

Operador	Associatividade
`()`	esquerda para direita
`!  -  ++  --  * & (cast)` (unários)	direita para esquerda
`*  /  %`	esquerda para direita
`+  -`	esquerda para direita
`<  <=  >  >=`	esquerda para direita
`==  !=`	esquerda para direita
`&&`	esquerda para direita
`||`	esquerda para direita
`=  +=  -=  *=  /=  %=`	direita para esquerda

Tabela 5: Precedência e associatividade de operadores

25 Entrada e Saída Padrão

A forma com que um programa em C se comunica com o mundo externo é através de entrada e saída de dados: o usuário fornece dados via teclado e o promrama imprime mensagens na tela. Todos os programas vistos até agora lêem suas entradas do teclado e produzem suas saídas na tela.

Em C toda entrada e saída é feita com *streams* (seqüências) de caracteres organizadas em linhas. Cada linha consiste de zero ou mais caracteres e termina com o caracter de final de linha. Pode haver até 254 caracteres em uma linha (incluindo o caracter de final de linha). Quando um programa inicia, o sistema operacional automaticamente define quem é a **entrada padrão** (geralmente o teclado) e quem é a **saída padrão** (geralmente a tela).

As facilidades de entrada e saída não fazem parte da linguagem C . O que existe é uma biblioteca padrão de funções para manipular a transferência de dadosentre programa e os dispositivos (*devices*) de saída e entrada padrão. Algumas destas funções são: `scanf()`, `printf()`, `getchar()`, `puts()`, `gets()`. Estas funções são declaradas no arquivo `<stdio.h>`. Existem funções úteis para conversão e teste de caracteres declaradas no arquivo `<ctype.h>`.

As funções de entrada e saída operam sobre *streams* de caracteres. Toda vez que uma função se entrada é chamada (por exemplo, `getchar()`, `scanf()`) ela verifica pela próxima entrada disponível na entrada padrão (por exemplo, texto digitado no teclado). Cada vez que uma função de saída é chamada, ela entrega o dado para a saída padrão (por exemplo, a tela).

As funções para leitura da entrada padrão e para escrita na saída padrão que têm sido usadas até agora são:

1. **Entrada e saída de caracteres:**

```
int getchar( void );
int putchar( int );
```

2. **Entrada e saída de *strings*:**

```
char *gets(char *);
int puts( char *);
```

3. **Entrada e saída formatada:**

```
int scanf(char *format, arg1, arg2, ... );
int printf(char *format, arg1, arg2, ... );
```

26 Arquivos

O armazenamento de dados em variáveis e *arrays* é temporário. Arquivos são usados para armazenamento permanente de grandes quantidades de dados (e programas) em dispositivos de armazenamento secundário, como discos.

Ás vezes não é suficeinte para um programa usar somente a entrada e saída padrão. Há casos em que um programa deve acessar arquivos. Por exemplo, se nós guardamos uma base de dados com endereços de pessoas em um arquivo, e queremos escrever um programa que permita ao usuário interativamente buscar, imprimir e mudar dados nesta base, este programa deve ser capaz de ler dados do arquivo e também gravar dados no mesmo arquivo.

No restante desta seção discutiremos como arquivos de texto são manipulados em C . Como será visto, tudo ocorre de maneira análoga ao que acontece com entrada e saída padrão.

26.1 Acessando um arquivo: `FILE *`, fopen(), fclose()

C visualiza cada arquivo simplesmente como um *stream* seqüencial de bytes. Da mesma forma que uma *string* em C termina com o caracter nulo, `'\0'`, cada arquivo em C termina com um marcador de final de arquivo (*end-of-file*), `EOF`.

As regras para acessar um arquivo são simples. Antes que um arquivo seja lido ou gravado, ele é *aberto*. Um arquivo é aberto em um *modo* que descreve como o arquivo será usado (por exemplo, para leitura, gravação ou ambos). Um arquivo aberto pode ser processado por funções da biblioteca padrão em C . Estas funções são similares às funções de biblioteca que lêem e escrevem de/para entrada/saída padrão. Quando um arquivo não é mais necessário ele deve ser *fechado*. Ao final da execução de um programa todos os arquivos abertos são automaticamente fechados. Existe um número máximo de arquivos que podem ser simultaneamente abertos de forma que você deve tentar fechar arquivos quando você não precisa mais deles.

Quando um arquivo está *aberto*, um *stream* é associado ao arquivo. Este *stream* fornece um canal de comunicação entre um arquivo e o programa. Três arquivos e seus respectivos *streams* são abertos automaticamente quando um programa inicia sua execução: a entrada padrão, a saída padrão e a saída padrão de erros.

A função da biblioteca padrão fopen() é usada para abrir um arquivo. fopen() toma dois argumentos do tipo *string*: o primeiro argumento é o nome do arquivo (por exemplo data.txt), o segundo argumento é a indicação do modo no qual o arquivo deve ser aberto. fopen() negocia com o sistema operacional e retorna um ponteiro para um tipo estrutura especial `FILE`. Este ponteiro é chamado *file pointer*, e aponta para uma estrutura que contém informações de sistema sobre o arquivo. O tipo `FILE` é predefinido em <stdio.h>. O *file pointer* é usado pelas funções de biblioteca que processam o arquivo aberto, e "representa"o arquivo do momento em que é aberto até o momento em que é fechado. A estrutura `FILE` é referida como *file control block* (FCB). Cada arquivo possui um FCB correspondente no disco. Quando um arquivo é aberto sua FCB é copiada para a memória e um ponteiro é definido para lá. O processamento do arquivo usa o ponteiro para o FCB para manipular arquivos, de forma que o tipo do ponteiro é `FILE *`. A saída padrão, a entrada padrão e a saída padrão de erros são manipulados usando ponteiros pré-definidos chamados `stdout, stdin` e `stderr` respectivamente.

O usuário não necessita saber detalhes de como a transferência de dados entre programa e arquivo é feita. As únicas sentenças necessárias no programa são a definição de uma variável do tipo `FILE *` (um *file pointer*) e uma atribuição de um valor para aquela variável por fopen(). As sentenças abaixo dão um exemplo de como abrir o arquivo data.txt para leitura.

```
FILE *fp;
fp = fopen("data.txt", "r");
```

O protótipo da função fopen() é:

```
FILE *fopen(char *name, char *mode);
```

fopen() recebe dois argumentos: o primeiro é uma *string* que é um nome de um arquivo a ser aberto, e o segundo é uma *string* que representa o modo de abertura do arquivo: `"r"` indica que o arquivo será aberto apenas para leitura, `"w"`, para escrita apenas. Se o arquivo não existe e é aberto para escrita, fopen() cria o arquivo. Se um arquivo já existente é aberto para escrita, o seu conteúdo é descartado. Há outros modos, incluindo anexação a um arquivo, leitura e escrita simultânea; para mais detalhes, veja a documentação da função nos livros-texto ou no manual on-line.

Se um arquivo é aberto com sucesso, o endereço da estrutura `FILE` é retornado por fopen(). Se a tentativa de abertura resulta em erro, fopen() retorna um ponteiro nulo, `NULL`. Alguns dos error possíveis são: abrir um arquivo que não existe para leitura, abrir um arquivo para escrita quando não há mais espaço disponível em disco, ou abrir um arquivo para qualquer operação sendo que as permissões de acesso do arquivo não o permitem.

É recomendável que você teste o valor de retorno de fopen() para verificar se houve erro de abertura. O trecho de programa abaixo ilustra como fazê-lo:

```
#include <stdio.h>
....
FILE *fp;
char fnome[13];
char fmodo[3];

printf("Entre um nome de arquivo para abrir:");
scanf("%s", fnome);
printf("Entre o modo de abertura do arquivo:");
scanf("%s", fmodo);

fp = fopen( fnome, fmodo );
if (fp == NULL)
   {
   printf("Erro na abertura de %s no modo %s\n", fnome, fmodo);
   return ;
   }
else
   printf("Arquivo %s aberto com sucesso no modo %s\n", fnome, fmodo);
...
```

No exemplo acima se o arquivo não puder ser aberto com sucesso, uma mensagem apropriada é exibida na saída padrão e o programa termina. Caso contrário uma mensagem indicando o sucesso na abertura do arquivo é exibida e o programa continua sua execução.

Cada arquivo aberto possui seu próprio *file pointer*. Por exemplo, se um programa vai manipular dois arquivos diferentes arq1 and arq2 simultaneamente (um para leitura e outro para escrita), dois *file pointers* devem ser usados:

```
FILE *fp1, *fp2;

fp1 = fopen("arq1", "r");
fp2 = fopen("arq2", "w");
```

Os valores de *file pointer* (`FILE *`) são chamados *streams*. Eles estabelecem conexão entre o programa e o arquivo aberto. A partir do momento de abertura, o nome do arquivo é irrelevante para o programa. Todas as funções que operam sobre o arquivo usam o *file pointer* associado.

Terminada a manipulação do arquivo o programa deve fechar o arquivo. A função padrão fclose() é usada com este propósito. Ela quebra a conexão entre o *file pointer* e o arquivo. Esta função toma como argumento o *file pointer* que representa o arquivo a ser fechado. file to be closed. O protótipo de fclose() é:

```
int fclose(FILE *);
```

fclose() retorna `0`, se há sucesso ou `EOF` em caso contrário. Abaixo um exemplo de uso de fclose():

```
fclose(pf1);
fclose(pf2);
```

26.2 Processando arquivos de texto

Arquivos podem guardar duas categorias básicas de dados: **texto** (caracteres no universo ASCII) ou **binário** (como dados armazenados em memória ou dados que representam uma imagem JPEG).

Depois que um arquivo de texto é aberto, existem 3 formas diferentes de ler ou escrever sequencialmente os dados: (i) um caracter por vez, usando as funções da biblioteca padrão fgetc() e fputc(); (ii) uma linha (*string*) por vez, usando fgets() e fputs(); e (iii) em um formato específico, usando fscanf() e fprintf().

Arquivos binários podem ser lidos como registros de dados estruturados. Além disso, uma vez que todos os registros tem o mesmo tamanho, os dados podem ser acessados de forma não-sequencial (acesso aleatório). As funções usadas para isto são fwrite() e fread().

Outras funções de entrada e saída de mais baixo nível que podem ser usadas são as funções read() e write(). Estas funções não serão usadas no momento e geralmente somente programadores experientes as usam.

26.2.1 Entrada e saída de caracteres

As funções fgetc() e putc() são similares a getchar() e putchar(). Elas operam sobre um arquivo aberto cujo *file pointer* é passado como argumento.

Os protótipos de fgetc() e fputc() are

```
int fgetc(FILE *fp);
int fputc(char ch, FILE *fp);
```

getc() returns the next character read from the file represented by the stream `fp`, or `EOF` if error or end of file occurs. putc() writes the character `ch` in the file represented by the stream `fp`. It returns the character written or `EOF`.

Abaixo segue um exemplo de programa que lê um arquivo caracter a caracter e imprime o que foi lido na saída padrão (a tela do computador):

```
/*************************************************************************
 * Lê um caracter por vez de um arquivo e
 * o imprime na saída padrão
 *************************************************************************/
#include <stdio.h> /* para funções padrão de E/S */

main()
{
   FILE *fp;
   char fnome[13];
   int ch;

   /* dialogo com usuário */
   printf("Entre um nome de arquivo: ");
   scanf("%s", fnome);

   fp = fopen( fnome, "r" ); /* abre arquivo*/
if (fp == NULL)
      {
      printf("Erro ao abrir %s\n", fnome);
      return;
      }
   else
      {
      printf("Arquivo aberto com sucesso.\n");

   /* Lê o arquivo caracter a caracter e imprime em stdout (saída padrão) */
```

```
   while( (ch=fgetc(fp)) != EOF )
          printf("%c", ch);

      fclose(fp); /* fecha arquivo */
}
}
```

26.2.2 Entrada e saída de *strings*

As funções fgets() and fputs() são similares a gets() e puts(). Elas operam sobre um arquivo aberto cujo *file pointer* é passado como argumento.

Os protótipos de fgets() e fputs() são:

```
char *fgets(char *str, int n, FILE *fp);
int fputs(char *str, FILE *fp);
```

A função fgets() lê do arquivo conectado ao *stream* `fp` no máximo `(n - 1)` caracteres para o *array* `str`, parando a leitura se o caracter `'\n'` (uma mudança de linha) é encontrado; O caracter `'\n'` é incluído no *array* e ao elemento do *array* seguinte é atribuído `'\0'` (final de *string*). A função fgets() retorna `str` ou o ponteiro nulo, `NULL`, se um erro ou final de arquivo ocorre.

A função fputs() escreve no arquivo conectado ao *stream* `fp` a *string* `str`, retornando um número não-negativo, ou `EOF` em caso de erro.

No exemplo a seguir é usada a função fgets() (e não gets(). Você pode dizer por quê?) para salvar em um arquivo um texto digitado através da entrada padrão (stdin). Para sinalizar pelo teclado que você terminou de entrar o texto, deve-se teclar `^D` (`^Z` Turbo C) e então a tecla ENTER.

```
/*********************************************************************
 * Escreve texto digitado em stdin em um arquivo
 *********************************************************************/
#include <stdio.h> /* funções padrão de E/S */

main()
{
   FILE *fp;
   char fnome[13];
   char linha[81];

   /* dialogo com usuario */
   printf("Entre um nome de arquivo: ");
   scanf("%s", fnome);

   fp = fopen( fnome, "w" ); /* abre arquivo. Conteúdo anterior é perdido.*/
 if (fp == NULL)
      {
      printf("Erro ao abrir %s\n", fnome);
      return;
      }
   else
      {
      printf("Arquivo aberto com sucesso");

      /* lê linha do teclado, armazena em uma string,
```

```
        * salva string em arquivo */

       while( fgets(linha, 80, stdin ) != NULL)
          fputs(linha, fp);
       fclose(fp); /* fecha arquivo */
}
}
```

26.2.3 Entrada e saída formatada: fscanf(), fprintf()

Para entrada e saída formatada as funções padrão fscanf() e fprintf() podem ser usadas. Elas são idênticas às funções scanf() e printf(), exceto que elas têm um argumento adicional (o primeiro em sua lista de argumentos) que é o *stream* conectado ao arquivo a ser lido ou escrito. Informalmente, seus protótipos podem ser escritos como:

```
int fscanf( FILE *fp, char *format, arg1, arg2, ... );
int fprintf(FILE *fp, char *format, arg1, arg2, ... );
```

A função fscanf() lê do arquivo representado pelo *stream* `fp` sob controle de um *string* de formato `format`. O *string* de formato geralmente contém conversões (como `%d, %s, %f`) que dirigem a interpretação da entrada. Os valores convertidos são atribuídos para os argumentos subsequentes, cada qual **devendo ser um ponteiro**. A função fscanf() retorna quando o *string* de formato foi totalmente interpretado. O valor retornado por fscanf() é `EOF` se o final do arquivo foi atingido ou um erro ocorre, caso contrário retorna a quantidade de itens convertidos e atribuídos.

A função fprintf() escreve no arquivo conectado ao *stream* `fp` sob controle de um *string* de formato `format`. O *string* de format contém dois tipos de objetos: caracteres ordinários que são copiados do jeito que são, e especificadores de conversão que causam a conversão e impressão dos argumentos seguintes de fprintf(). O valor de retorno é o número de caracteres escritos, ou negativo em caso de ocorrência de erros.

Abaixo segue um exemplo simples de base de dados. Os dados são armazenados permanentemente em um arquivo, agentes.txt. A base de dados contém registros de agentes secretos famosos. Para cada agente um apelido e um número de código são armazenados. Uma vez que será usada e/s formatada, deve-se conhecer o formato no qual os dados estão armazenados no arquivo da base de dados. Este formato é: os dados de diferentes agentes estão em linhas separadas; para cada agente em uma linha, tem-se primeiro o apelido e então o código numérico, separados por espaço.

O programa orientado a menu abaixo lista todos os registros e adiciona novos itens.

```
/*************************************************************
 * programa com menu para operar uma base de dados de no máximo 50
 * agentes secretos; a base de dados é guardada permanentemente em
 * um arquivo em disco.
 *************************************************************/

#include <stdio.h>                    /* funções padrão de E/S */
#define FNOME "agentes.txt"             /* nome do arquivo de dados */
#define NUM 50                        /* numero de registros na base de dados */
#define NOMELEN 30                    /* tamanho de um nome */

/*** declara estrutura de dados ***/
struct pessoal
   {
   char nome [NOMELEN];          /* nome codigo(sem espaços em branco) */
   int agnum ;                   /* numero codigo */
```

```
   };

/*** prototipos ***/
int  cargadb(struct pessoal []);
int  novonome(struct pessoal [], int);
void listatudo(struct pessoal [], int);
void salvadb(struct pessoal [], int);
```

```
/****** MAIN *********/
main()
{
   struct pessoal agentes[50];     /* array de 50 estruturas      */
   int n;                          /* indice para o ultimo registro ativo */
   char ch;

   /*** carrega a base de dados em agentes[], n é o tamanho da base de dados
   n = loaddb(agentes);

   /* seleciona uma opção do menu e processa os dados em memória  */
   do {
      printf("\nDigite 'e' para entrar novo agente,");
      printf("\n     'l' para listar todos os agentes,");
      printf("\n     'q' para terminar: ");
      ch = getchar();
      switch (ch)
         {
         case 'e':
            n = novonome(agentes, n);   /* adiciona um novo agente no indice n
            break;
         case 'l':                      /* lista todos os registros */
            listatudo(agentes, n);
            break;
         case 'q':                      /* salva todos os registros */
            salvadb(agentes, n);
            break;
         default:                       /* Engano do usuario  */
            printf("\nEntre somente as opções listadas.\n");
         }
         while (fgetc(stdin) != '\n') ;
      } while (ch != 'q');
}
```

Uma amostra de uma execução do programa segue abaixo. Inicialmente o conteúdo do arquivo agentes.txt é:

```
Klara 89
Edward 888
ZipZap 109
```

Uma amostra de execução:

```
Digite 'e' para entrar novo agente,
       'l' para listar todos os agentes,
       'q' para terminar: l
Klara 89
Edward 888
ZipZap 109

Digite 'e' para entrar novo agente,
       'l' para listar todos os agentes,
       'q' para terminar: e
Digite nome e código: TipTop 999

Digite 'e' para entrar novo agente,
       'l' para listar todos os agentes,
       'q' para terminar: l
Klara 89
Edward 888
ZipZap 109
TipTop 999

Digite 'e' para entrar novo agente,
       'l' para listar todos os agentes,
       'q' para terminar: q
Salvar? ('s' para salvar)
y
Salvando...Feito
```

A seguir, apresenta-se a implementação das quatro funções cargadb(), novonome(), listatudo(), and salvadb().

```
/*********************************************************************
 * lê a base de dados do arquivo (até EOF) no array em memória
 * ENTRADA: um array de do tipo struct pessoal
 * RETORNO: número de elementos lidos
 * SUPOSIÇÕES: o tamanho da base de dados deve ter no máximo 50 registros
 *********************************************************************/
int cargadb(struct pessoal pessoa[])
{
   int i = 0;
   FILE *fp;                    /* define ptr to FILE          */

   fp = fopen(FNOME, "r");

   while ( fscanf(fp, "%s %d", pessoa[i].nome, &pessoa[i].agnum) != EOF )
      i++;
   fclose(fp);
   return i;
}
```

```
/*********************************************************************
 * adiciona novo elemento ao indice n no array pessoa[],
 * o valo da estrutura é obtido da entrada padrão
 * ENTRADA: array pessoa[] -- to store the structure value
 *                       n -- indice do elemento, incrementado a
 *                            cada novo elemento
 *********************************************************************/
int novonome(struct pessoal pessoa[], int n)
   {
      if (n < NUM)
      {
         printf("Digite nome e código: ");
         scanf("%s %d", pessoa[n].nome, &pessoa[n].agnum);
         n++;
      }
      else
         printf("Não há mais espaço\n");
      return n;
   }

/*********************************************************************
 * imprime a base de dados na tela
 * ENTRADA: array pessoa[] a imprimir
 *                       n número de registros para imprimir
 *********************************************************************/
void listatudo(struct pessoal pessoa[], int n)
   {
   int j;

   for (j = 0; j < n; j++)
       {
       printf("%s %d\n", pessoa[j].nome, pessoa[j].agnum);
       }
   }

/*********************************************************************
 * Pergunta ao usuário se quer salvar a base de dados. Se a resposta é SIM (s)
 * abre o arquivo para escrita e grava o array no arquivo
 * ENTRADA: array pessoa[] a ser salvo
 *                       n número de registros a ser salvo
 *********************************************************************/
void salvadb(struct pessoal pessoa[], int n)
{
  int i;
  FILE *fp;

  while (fgetc(stdin) != '\n') ;
  printf("Salvar? ('s' para salvar)\n");
  if ( getchar() == 's')
     {
```

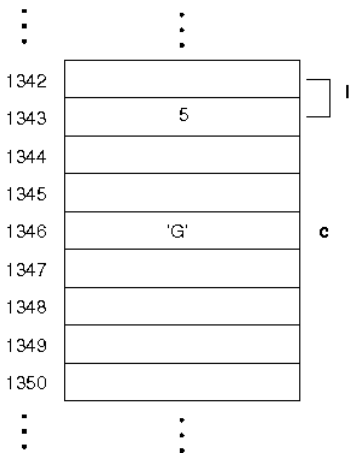
```
      fp = fopen(FNOME, "w");
      printf("Salvando...");
      for (i = 0; i < n; i++)
        {
        fprintf(fp, "%s %d\n", pessoa[i].nome, pessoa[i].agnum);
        }
      fclose(fp);
      printf("Feito.\n");
      }
  else
      printf("Alterações não foram salvas.\n");
}
```

27 Ponteiros

Em linguagem C a cada variável está associado: (i) um nome; (ii) um tipo; (iii) um valor; e (iv) um endereço. Considere as seguintes definições de variáveis.

```
int i = 5;
char c = 'G';
```

Na memória, eles podem estar armazenados da forma abaixo:



A variável inteira `i` está armazenada no endereço `1342`. Ela usa dois bytes de memória (quando um objeto usa mais de um byte, seu endereço é onde ele começa – neste caso, `1342` e não `1343`). A variável do tipo `char c` está armazenada no endereço `1346` e usa um byte de memória. O compilador é que controla do local de armazenamento destas variáveis em memória.

27.1 O operador de endereço (`&`)

Nós podemos usar o operador de endereço para determinar o endereço de uma objeto na memória. Este operador só pode ser usado com lvalues (objetos que podem estar no lado esquerdo de uma atribuição, como no caso de variáveis) porque lvalues tem um endereço alocado na memória.

Por exemplo, no exemplo acima, poderíamos usar o operador de endreço como nas expressões abaixo:

`&i` tem valor `1342`
`&c` tem valor `1346`

27.2 Tipo ponteiro

Em C, uma variável que contém um endereço de memória é uma variável do tipo **ponteiro**. Um valor, que é um endereço (como `&a`) é um valor de ponteiro. Quando um ponteiro (a variável) contém um determinado endereço, dizemos que ele *aponta* para o endereço de memória. Além disso, se o valor deste ponteiro é o endereço de uma outra variável qualquer, dizemos que tal ponteiro *aponta* para esta outra variável.

Há um tipo distinto de ponteiro para cada tipo básico C (como `int`, `char` e `float`). É verdade que todos os endereços tem o mesmo tamanho[4], mas nós também precisamos saber algo sobre o que é armazenado no endereço de memória apontado (quantos bytes ocupa e como os bytes devem ser interpretados).

Assim, a declaração de um tipo ponteiro em C é feita da seguinte forma:

```
tipo *nome_var;
```

[4]O operador sizeof() pode ser usado para determinar o tamanho de um ponteiro. Por exemplo, sizeof (char *)

Esta declaração indica que está sendo definido um ponteiro para tipo chamado nome_var. Por exemplo, um tipo ponteiro usado para *apontar* para inteiros é chamado *ponteiro para int* e isso é denotado por um `int *`. Variáveis do tipo ponteiro para `int` são usadas para armazenar endereços de memória que contem valores do tipo `int`.

Dadas as definições de `i` e `c` acima, nós podemos definir duas novas variáveis `pi` e `pc`, ambos do tipo ponteiro.

```
int *pi;
char *pc;
```

Nesta definição as variáveis não foram inicializadas com nenhum valor. Podemos inicializá-las com:

```
pi = &i;
pc = &c;
```

Depois destas atribuições, o valor de `pi` seria `1342`, e o valor de `pc` seria `1346`.

Note que nesta definição da variável `int *pi`, `pi` é o nome da variável e `int *` é o tipo de `pi` (ponteiro para `int`).

27.3 O operador de dereferência: *

Quando um ponteiro aponta para um endereço de memória, a operação para acessar o conteúdo do endereço apontado é chamado de **dereferência**. O operador unário `*` é usado para fazer a dereferência. Note que este uso do símbolo `*` não tem relação com o símbolo de multiplicação. Usando os exemplos anteriores, `*pi` é o objeto apontado por `pi` (no caso, o valor de um inteiro).

`*pi` tem valor `5`
`*pc` tem valor `'G'`

Como um ponteiro dereferenciado (tais como `*pi` ou `*pc`) refere-se a um objeto na memória, ele pode ser usado não só como valor, mas também como um lvalue. Isto significa que um ponteiro dereferenciado pode ser usado no lado esquerdo de uma atribuição. Veja alguns exemplos:

```
printf("Valor= %d, Char = %c\n", *pi, *pc);
*pi = *pi + 5;
*pc = 'H';
```

`*pi` no lado esquerdo do = refere-se ao endereço de memória para o qual `pi` aponta. `*pi` no lado direito do = refere-se ao valor armazenado no endereço apontado por `pi`. A sentença `*pi = *pi + 5;` faz com que o valor armazenado no endereço apontado por `pi` seja incrementado de `5`. Note que o valor de `*pi` muda, não o valor de `pi`.

Neste exemplo, os valores das variáveis `i` e `c` poderiam ter sido alterados sem a utilização de ponteiros da seguinte forma:

```
printf("Valor = %d, Char = %c\n", i, c);
i = i + 5;
c = 'H';
```

Os exemplos acima ilustram como uma variável pode ser acessada diretamente (através do seu nome) ou indiretamente (através de um ponteiro apontando para o endereço da variável).

27.4 Atribuições envolvendo ponteiros

Um ponteiro pode ter atribuído a si um valor que seja o endereço de memória onde está armazenado um valor do mesmo tipo do ponteiro. Isto ocorre quando se usa o operador de endereço visto acima, ou quando se usa o valor de um outro ponteiro que aponte para um objeto do mesmo tipo do primeiro ponteiro. Observe-se o exemplo abaixo:

```
int *p1, *p2, x;
float *p3;

p1 = &x;        /* Correto */
p2 = p1;        /* Correto */
p3 = p1;        /* Incorreto. Compilador acusa "Warning". */
```

No exemplo acima, a linguagem C admite a atribuição de um ponteiro para outro de outro tipo (`p3 = p1;`), mas a compilação acusa uma mensagem de aviso. Posteriormente serão vistas situações em que a atribuição de ponteiros de tipos diferentes devem ocorrer e como devem ser manipuladas em C.

27.5 Aritmética de ponteiros

Apenas as operações de adição e subtração (e operadores C associados) são permitidos com ponteiros. Assim, é possível adicionar ou subtrair valores inteiros de ponteiros.

Operações de soma, subtração e comparação entre ponteiros também são válidas, desde que os ponteiros envolvidos apontem para o mesmo tipo de dados. Ainda assim, o resultado somente terá algum sentido prático se os ponteiros apontarem também para o mesmo objeto.

Alguns exemplos:

```
int num[20], *pnum, diff;
char str[30], *pstr, *pn, char nome[20];

pn = nome;
pstr = str;
pnum = num;

pnum += 3;              /* pnum = &num[3] */
*pnum = 10;             /* equivale a num[3] = 10 */

pstr++;                 /* pstr = &str[1] */

diff = pstr - pnum;     /* INCORRETO. Os ponteiros apontam para
                         * tipos diferentes
                         */
diff = pstr - pn;       /* CORRETO, mas o valor não tem
                         * necessáriamente o sentido de "numero
                         * de bytes entre pn e pstr".
                         */
pn = str;
pstr = &str[30];

diff = pstr - pn;       /* CONCEITUALMENTE CORRETO. diff == 30 */
```

Um último ponto a respeito de operações sobre ponteiros: Adicionar um ponteiro a outro não produz nenhum resultado prático ou válido.

27.6 Ponteiros e *Arrays*

Em C, o nome de uma variável que foi declarada como *array* representa um ponteiro que aponta para o início do espaço de armazenamento do *array*, isto é, o endereço de memória do primeiro byte associado ao primeiro elemento do array:

```
char nome[20],
     *pstr;
int val[10],
    *ptr;


pstr = nome;        /* Equivalente a pstr = &nome[0] */
ptr = val;          /* Equivalente a ptr = &val[0] */

pstr = nome + 4;    /* Equivalente a pstr = &nome[4] */
ptr = val + 5;      /* Equivalente a ptr = &val[5] */

pstr = nome++;      /* ATENCAO: INCORRETO !!! */
                    /* "nome" NÃO É UM PONTEIRO */
```

Se um ponteiro aponta para um *array*, pode-se usar indistintamente as formas abaixo para acessar os elementos do *array*:

```
int val[10],
    x,
    *ptr;

ptr = val;          /* Equivalente a ptr = &val[0] */

*(ptr + 3) = 7;    /* val[3] = 7 */
ptr[3] = 10;       /* val[3] = 10 */
                   /* Equivalente a *(ptr + 3) = 10 */

ptr += 4;
ptr[3] = 20;       /* ATENCAO:  val[7] = 20 */
```

27.7 Ponteiros e Estruturas

Como em qualquer outro tipo, ponteiros para estruturas podem ser definidos. Considere o exemplo abaixo:

```
/* declara uma estrutura */
struct facil {
   int num;
   char ch;
};

main()
{
    /* definioes de variaveis */
    struct facil fac,      /* uma variavel do tipo "struct facil" */
                 *pfac;    /* um ponteiro para "struct facil" */

    pfac = &fac;

    (*pfac).num = 32; /* o membro "num" da "struct facil" apontada por "pfac
    (*pfac).ch = 'A'; /* o membro "char" da "struct facil" apontada por "pfa
  }
```

Como se espera, quando se usa um ponteiro para um tipo `struct`, o ponteiro deve ter assinalado a si um valor ANTES de ser dereferenciado. A ordem pela qual um membro pode ser acessado através do ponteiro, the pointer é: primeiro o ponteiro é dereferenciado, e então o operador de membro de estrutura e o nome do membro são usados para acessar um membro em particular da estrutura apontada pelo ponteiro. Uma vez que o operador `.` tem precedência mais alta que o operador `*` (veja Tabela 6), os parenteses são necessários.

27.7.1 Acesso a membros de estrutura via ponteiro: O operador ->

Uma notação do tipo `(*pfac).ch` é confusa, de forma que a linguagem C define um operador adicional (->) para acessar membros de estruturas através de ponteiros. O operador -> é formalmente usado como o operador `.`, exceto que ao invés do nome da variável de estrutura, um ponteiro para o tipo `struct` é usado à esquerda do operador ->. No exemplo acima, as duas últimas linhas de código podem portanto ser reescritas como:

```
pfac->num = 32;    /* o mesmo que (*pfac).num = 32;  */
pfac->ch = 'A';    /* o mesmo que (*pfac).ch = 'A';  */
```

Basicamente, use o operador `.` se você tem uma variável de tipo `struct`, e o operador -> caso você tenha um ponteiro para um tipo `struct`.

27.8 Ponteiros como argumentos de funções

Nos exemplos acima, pode parecer que ponteiros não são úteis, já que tudo que fizemos pode ser feito sem usar ponteiros. Agora, considere o exemplo da função `troca()` abaixo, que deve trocar os valores entre seus argumentos:

```
#include <stdio.h>

void troca(int, int);

void troca(int x, int y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}

main(void)
{
    int a, b;

    printf("Entre dois numeros: ");
    scanf("%d %d", &a, &b);

    printf("Voce entrou com %d e %d\n", a, b);
```

```
    /* Troca a e b */
    troca(a, b);

    printf("Trocados, eles sao %d e %d\n", a, b);
}
```

Quando `a` e `b` são passados como argumentos para `troca()`, na verdade, somente seus valores são passados. A função não pode alterar os valores de `a` e `b` porque ela não conhece os endereços de `a` e `b`. Mas se ponteiros para `a` e `b` forem passados como argumentos ao invés de `a` e `b`, a função `troca()` seria capaz de alterar seus valores; ela saberia então em que endereço de memória escrever. Na verdade, a função não sabe que os endereços de memória são associados com `a` e `b`, mas ela pode modificar o conteúdo destes endereços. Portanto, passando um ponteiro para uma variável (ao invés do valor da variável), habilitamos a função a alterar o conteúdo destas variáveis na função chamadora.

Uma vez que endereços de variáveis são do tipo ponteiro, a lista de parâmetros formais da função deve refletir isso. A definição da função `troca()` deveria ser alterada, e a lista de parâmetros formais deve ter argumentos não do tipo `int`, mas ponteiros para `int`, ou seja, `int *`. Quando chamamos a função `troca()`, nós não passamos como parâmetros reais `a` e `b`, que são do tipo `int`, mas `&a` e `&b`, que são do tipo `int *`. Dentro da função `troca()` deverá haver mudanças também. Uma vez que agora os parâmetros formais são ponteiros, o operador de dereferência, `*`, deve ser usado para acessar os objetos. Assim, a função `troca()` é capaz de alterar os valores de `a` e `b` "remotamente".

O programa abaixo é a versão correta do problema enunciado para a função `troca()`:

```
#include <stdio.h>

void troca(int *, int *);

/* function troca(px, py)
 *  acao:        troca os valores inteiros apontados por px e py
 *  entrada:     apontadores px e py
 *  saida:       valor de  *px e *py trocados
 *  suposicoes:  px e py sao apontadores validos
 *  algoritmo:   primeiro guarda o primeiro valor em um temporario e troca
 */
void troca(int *px, int *py)
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}

main(void)
{
    int a, b;

    printf("Entre dois numeros: ");
    scanf("%d %d", &a, &b);

    printf("Voce entrou com %d e %d\n", a, b);
```

```
    /* Troca a e b -- passa enderecos */
    troca(&a, &b);

    printf("Trocados, eles sao %d e %d\n", a, b);
}
```

A saída deste programa é:

```
Entre dois numeros: 3 5
Voce entrou com  3 e 5
Trocados, eles sao 5 e 3
```

Basicamente, se a função precisa alterar o valor de uma variável na função chamadora, então passamos o endereço da variável como parâmetro real, e escrevemos a função de acordo, ou seja, com um ponteiro como parâmetro formal.

27.8.1 *Arrays* como argumentos de funções

Quando um *array* é passado como argumento para uma função, somente o ponteiro para a primeira posição do *array* é passada e não o conteúdo de todo o *array*. *Arrays* são portanto passados por referência e não por valor.

Ao se definir um *array* como o argumento formal de uma função em C, duas formas podem ser usadas. Elas podem ser vistas abaixo nas definições das funções `func_1()` e `func_2()`.

```
func_1 (char vet [], int ivet[])
{
   vet[3] = 'A';
   vet++;
   ivet += 3;
}

func_2 (char *vet, int *ivet)
{
   vet[4] = 'B';
   vet++;
   ivet += 3;
}

main()
{
  char ender[20];
  char vals[20];

  func_1(ender, vals);
  func_2(ender, vals);
}
```

Observe no exemplo acima que a passagem dos *arrays* ao se chamar as funções `func_1()` e `func_2()` é feita da mesma forma: Usa-se o NOME das variáveis declaradas como *arrays*. Note também o uso dos argumentos formais nas funções: `vet` e `ivet` podem ser usadas como ponteiros ou como nomes de *arrays* (com a notação indexada por []).

27.8.2 Ponteiros para estruturas como argumentos de funções

Quando estruturas são passadas como argumentos para funções o valor de todo o objeto agregado é passado literalmente. Além disso, se este valor é alterado na função, ele deve ser retornado (via `return`), o que implica em copiar de volta toda a estrutura. Isto pode ser bastante ineficiente no caso de uma estrutura grande (com muitos membros, com membros de tamanho grande como *arrays*, etc.). Assim, em alguns casos é melhor passar ponteiros para estruturas. Repare a diferença com *arrays* passados como argumentos para funções vista na seção anterior.

O programa abaixo é um exemplo do uso de passagem de ponteiros de estruturas para funções:

```
#define LEN 50

struct endereco {
    char rua[LEN];
    char cidade_estado_cep[LEN];
};

void obtem_endereco(struct endereco *);
void imprime_endereco(struct endereco);

void obtem_endereco(struct endereco *pender)
{
     printf("Entre rua: ");
     gets(pender->rua);
     printf("Entre cidade/estado/cep: ");
     gets(pender->cidade_estado_cep);
}

void imprime_endereco(struct endereco ender)
{
     printf("%s\n", ender.rua);
     printf("%s\n", ender.cidade_estado_cep);
}

main()
{
     struct endereco residencia;

     printf("Entre seu endereco residencial:\n");
     obtem_endereco(&residencia);

     printf("\nSeu endereco:\n");
     imprime_endereco(residencia);
}
```

Neste caso, `main()` passa para a função `obtem_endereco()` um ponteiro para a variavel `residencia`. `obtem_endereco()` pode então alterar o valor de `residencia` "remotamente". Este valor, em `main()`, é então passado para `imprime_endereco()`. Note-se que não é necessário passar um ponteiro para a estrutura se seu valor não será mudado (como é o caso da função `imprime_endereco()`).

De um modo geral, é melhor passar ponteiros para estruturas ao invés de passar e retornar valores de estruturas. Embora as duas abordagens sejam equivalentes e funcionem, o programa irá apenas passar

ponteiros ao invés de toda uma estrutura que pode ser particularmente grande, implicando em um tempo final de processamento maior.

27.9 Precedência de operadores

A precedência dos operadores `*` e `&` é alta, a mesma que outros operadores unários. A tabela 6 apresenta a precedência de todos os operadores vistos até agora.

Operador	Associatividade
`()  []  ->  .`	esquerda para direita
`!  -  ++  --  * & (cast)` (unários)	direita para esquerda
`*  /  %`	esquerda para direita
`+  -`	esquerda para direita
`<  <=  >  >=`	esquerda para direita
`==  !=`	esquerda para direita
`&&`	esquerda para direita
`||`	esquerda para direita
`=  +=  -=  *=  /=  %=`	direita para esquerda
`,`	esquerda para direita

Tabela 6: Precedência e associatividade de operadores

28 *Strings* e Ponteiros

Vimos anteriormente que *strings* em C nada mais são que *arrays* de elementos do tipo `char` cujo último elemento é o caracter '\0' (nulo). Até agora a declaração e inicialização de um *string* tem sido como abaixo:

```
char nome[] = "Um string de caracteres";
char str[20] = "Um outro string de caracteres";
```

No exemplo acima, cada conjunto de caracteres limitados por um par de aspas é denominado de uma **constante *string***. Como toda expressão em C, uma constante *string* tem um valor e um tipo associado.

O compilador C automaticamente aloca espaço no programa para armazenar este *string*. Esta constante na verdade está definindo um *array* que está armazenado em um endereço de memória mas que não têm um nome de variável associado a ele. O valor produzido para esta constante *string* é um ponteiro para este *array* anônimo. E o tipo da constante é um ponteiro para `char`.

Assim, na maior parte dos casos de programas em C, é mais conveniente representar um *string* através de um ponteiro para `char`. O valor inicial deste ponteiro deve ser sempre o endereço do primeiro caracter do *string* ou uma constante *string*. O exemplo acima pode ser reescrito como abaixo, que deve ser a forma preferida para se declarar e inicializar *strings* em programas em linguagem C:

```
char *nome = "Um string de caracteres";
char *str = "Um outro string de caracteres";
```

29 *Arrays* de ponteiros

Da mesma forma que se pode ter *arrays* de tipos básicos (e.g. `int`, `char`) e de estruturas, pode-se também definir *arrays* de ponteiros:

```
char *frases[60];

frases[0] = "Um string de caracteres";
frases[3] = "Um outro string de caracteres";
frases[10] = "Mais uma frase muito mais comprida que as outras acima";
frases[20] = "Agora uma frasesita pequena";
frases[30] = "Chega de frases";

puts(frases[3]);
```

No exemplo acima cada elemento do *array* `frase` é um ponteiro para `char`. Em cada uma das atribuições, cada elemento do *array* recebe como valor um ponteiro para um *string* (veja seção anterior sobre constantes *string*).

30 Argumentos de linha de comando: a função `main`

Uma aplicação de *arrays* de ponteiros surge de imediato quando você deseja acessar os argumentos digitados para seu programa na linha de comando.

Para isto, a função `main` manipula dois argumentos quando o programa inicia a execução:

```
main (int argc, char *argv[])
{
   .....
}
```

O primeiro argumento, denominado `argc` por convenção, representa o número de argumentos digitados na linha de comando, incluindo o nome do programa na linha de comando, que é definido como o primeiro argumento. Desta forma o valor de `argc` é sempre maior ou igual a 1 (um).

Em UNIX, um argumento é considerado como uma sequência de caracteres até a primeira ocorrência do caracter de espaçamento, que pode ser um espaço em branco, tab ou o caracter de mudança de linha[5].

O segundo argumento de `main` é o *array* de ponteiros para caracteres chamado `argv`, por convenção. O primeiro ponteiro em `argv`, `argv[0]`, aponta para o nome do programa em execução. Os elementos sucessivos do *array*, `argv[1]`, `argv[2]`, ..., `argv[argc - 1]`, contém ponteiros para os argumentos digitados na linha de comando após o nome do programa.

Como um exemplo, considere a execução do programa chamado `nroff`. Se a linha de comando abaixo é digitada:

```
nroff -mm -TXR memo1
```

então o valor de `argc` será 4 (o nome do programa mais os três argumentos que se seguem), e o argumento `argv` será um *array* de ponteiros para caracteres. O primeiro elemento deste *array* aponta para o *string* **"nroff"**, o segundo para **-mm"**, o terceiro para **-TXR"** e finalmente o quarto elemento aponta para o *string* **"memo1"**. Isto pode ser melhor visualizado na Figura 3.

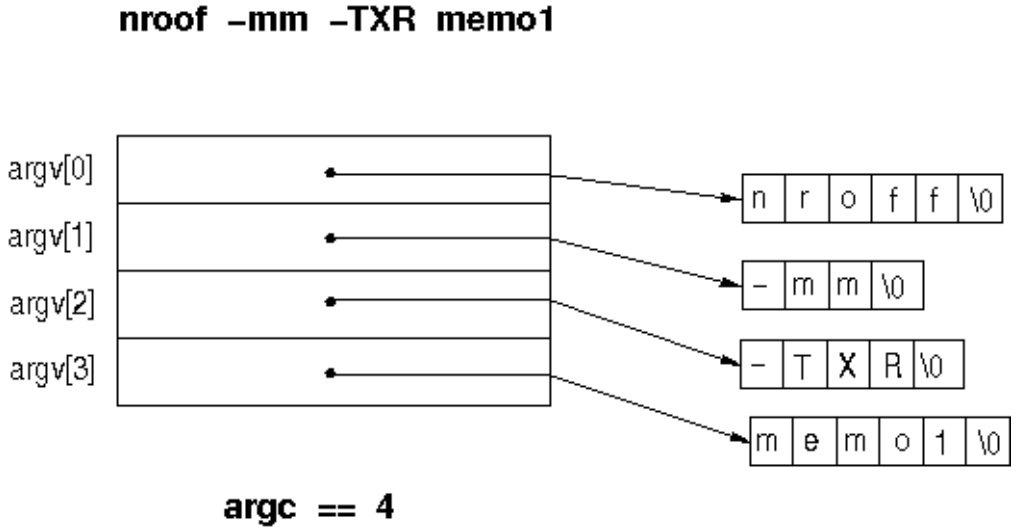


Figura 3: Argumentos na linha de comando

31 Alocação dinâmica de memória

Quando você declara um *array* em um programa C, você deve informar quantos elementos devem ser reservados para este *array*. Se você conhece este número *a priori*, tudo está bem.

No entanto, o tamanho de um *array* pode ser desconhecido por uma série de fatores. Por exemplo, se você deseja ler todas as linhas de um arquivo e armazená-los em um *array* de seu programa, o tamanho de memória necessário dependerá do tamanho do arquivo. E se este tamanho variar muito de um arquivo para outro, você terá que reservar espaço suficiente para acomodar o maior tamanho de arquivo possível, o que é um desperdício se seu programa vai lidar com muitos arquivos de tamanho reduzido e apenas alguns arquivos com tamanho grande.

Definir um tamanho máximo para o suas estruturas de dados aumenta o tamanho de seu programa. Em um ambiente multitarefa como UNIXeste seu programa estará competingo por espaço livre de memória. Se seu programa aloca espaço desnecessariamente, isto quer dizer que menos processos poderão ocupar

[5]Aspas e apóstrofes podem agrupar palavras como um argumento único, mesmo que tenham espaçamento.

a memória para serem executados. Portanto, durante a gerência do processo de sua tarefa pelo sistema operacional, a trasnferência do programa do disco para a memória e vice-versa (processo conhecido como *swapping*), irá demorar mais.

Para permitir que o espaço para estruturas de dados de um programa possam ser alocados durante a execução do mesmo é que a linguagem C define funções de biblioteca chamadas **funções de alocação dinâmica de memória**.

Estas funções são:

```
#include <stdlib.h>

void *malloc (int tamanho)
void *calloc (int numero_elementos, int tamanho_elemento)
void *realloc (void *area_alocada, int novo_tamanho)
void free (void *area_alocada)
```

As funções `malloc()` e `calloc()` são usadas para alocar espaço para seus dados uma vez que você determinou quanto espaço você precisa. E se sua estimativa se revela muito grande ou muito pequena, você pode mudar o tamanho deste espaço alocado com a função `realloc`. Finalmente, uma vez que o programa terminou de usar o espaço alocado, a função `free` pode ser usada para liberar o espaço previamente alocado. Observe-se a diretiva **#include** que necessária para o uso das funções de alocação.

31.1 `malloc` e `calloc`

As funções `malloc` e `calloc` alocam espaço de memória. `malloc` recebe como argumento o número de bytes a ser alocado. `calloc` recebe como primeiro argumento o número de elementos a aser alocado. O segundo argumento indica o tamanho em bytes de cada elemento. `calloc` garante que o espaço alocado é inicializado com zeros, enquanto que `malloc` não.

Estas duas funções retornam um ponteiro para a nova área alocada. O tipo do ponteiro por convenção é `char *`. Caso o ponteiro seja usado para apontar para outro tipo qualquer, um *cast* no valor retornado deve ser usado. Observe o exemplo abaixo:

```
#include <stdlib.h>

char buf[30],
     *memchar;
int *memint, i;

....

memchar = malloc (sizeof(buf));

if (memchar == NULL) {
   printf ("Erro em alocacao de memoria\n");
   exit (1);
}
else {
    memcpy (memchar, buf, sizeof(buf));  /* copia o conteúdo de
                                          * buf para memchar
                                          */
}

memint = (int *) calloc (1, sizeof(int));
```

```
if (memint == NULL) {
   printf ("Erro em alocacao de inteiros\n");
   exit (1);
}
else {
      i = 0;
      do {
          scanf("%d", memint + i);

          if (memint[i]) {
            memint = (int *) realloc (memint, (i + 2) * sizeof(int));

            if (memint == NULL)
               break;
          }
      } while (memint[i++]);
}
```

Oberve no exemplo acima o uso de *cast* na chamada de `calloc`. Como o ponteiro retornado será usado para apontar para inteiros, o retorno da função deve ser convertido de acordo com o *cast*.

Quando não há sucesso na alocação de memória, as funções `malloc` e `calloc` retornam um ponteiro nulo, representado pela constante **NULL**. Assim, toda vez que um programa usa estas funções, deve-se testar o valor retornado para verificar se houve sucesso na alocação. É o que acontece no exemplo acima ao se testar os valores de `memchar` e `memint` imediatamente após a chamada das funções de alocação.

31.2 `realloc`

A função `realloc` é usada para redimensionar um espaço alocado previamente com `malloc` ou `calloc`. Seus argumentos são um ponteiro para o INÍCIO de uma área previamente alocada, e o novo tamanho, que pode ser maior ou menor que o tamanho original.

`realloc` retorna um ponteiro para a nova área alocada. Este ponteiro pode ser igual ao ponteiro original se o novo tamanho for menor que o tamanho original, e diferente do ponteiro original se o novo tamanho for maior que o tamanho original. Neste último caso, `realloc` copia os dados da área original para a nova área.

O último bloco `else` no exemplo da seção anterior é um exemplo de uso da função `realloc`. Neste bloco, usa-se a função`realloc` para aumentar de 1 (um) o tamanho do *array* dinâmico representado por `memint`.

Referências

[1] Allen I. Holub. *Enough Rope to Shoot Yourself in the Foot: rules for C and C++ programming*. McGraw-Hill, 1995.