Curso de Linguagem C

Rodolfo Jardim de Azevedo

Instituto de Computação

UNICAMP

Programação Estruturada

- *A linguagem C é uma linguagem estruturada em bloco simples. Uma característica distintiva de uma linguagem estruturada em bloco é a compartimentalização de seu código e de seus dados, que é a habilidade de uma linguagem tem de seccionar e esconder do resto do programa todas as instruções necessárias para a realização de uma determinada tarefa.*

Declaração de variáveis

- *Variáveis devem ser declaradas antes de serem usadas, permitindo assim, que o compilador saiba de antemão informações como tipo e espaço gasto em memória podendo fazer checagem durante o processo de compilação.*

Funções - Blocos de código

- Utilizando funções, é possível esconder parte do código e variáveis, evitando assim, que sejam gerados efeitos colaterais em outras partes do programa. Desta forma, é necessário saber apenas o que as rotinas fazem, e não como elas fazem.

Laços

- *Os programas passam a maior parte do tempo repetindo tarefas até que uma condição seja satisfeita (ou um número fixo de vezes). Desta forma, fica mais fácil a programação e eliminam-se os inconvenientes gerados por vários gotos espalhados pelo programa.*

Testes de condições

- *Numa linguagem estruturada, os testes de condições são amplamente utilizados, tanto como controle de laços, quanto para execuções condicionais de blocos de código.*

A linguagem C

- *Surgiu nos anos 70 de uma linguagem chamada B. Criada por Dennis Ritchie. Embora houvessem poucas divergências entre as primeiras implementações em nível de código-fonte, foi desenvolvido o padrão ANSI sendo assim, qualquer programa C ANSI pode ser compilado em qualquer compilador C ANSI não importando a máquina na qual o programa vá ser executado. Por isso, quando se quer portabilidade, a escolha acaba recaindo sobre a linguagem C.*

Linguagem feita para programadores

- *Ao contrário do que possa parecer, nem todas as linguagens foram feitas para programadores. C é virtualmente única, porque ela foi criada, influenciada e testada em campo por programadores. Ela oferece ao programador exatamente o que ele quer: poucas restrições e queixas, código rápido e eficiência. Por isso ela é a linguagem mais popular entre os programadores profissionais altamente qualificados.*

Sensível ao caso

- *A linguagem C é sensível ao caso, isto quer significa que letras maiúsculas e minúsculas são tratadas como caracteres separados.*

Programa exemplo

```
/* Este é o primeiro programa */
#include <stdio.h>
main()
{
    printf("Alo mundo\n");
}
```

- Este é um exemplo bem simples, mas que mostra alguns componentes básicos que existem nos programas feitos em C. Nele vemos:

/* Este é o primeiro programa */

- Esta linha é um comentário.

#include <stdio.h>

- *Indica a inclusão de arquivos dentro do programa atual (neste caso, o arquivo stdio.h). Normalmente são arquivos cabeçalhos contendo declarações de tipos e protótipos de funções. Serão vistos mais tarde.*

main()

- *Todo programa em C tem que ter a função main, é na primeira linha desta função que o programa começa a ser executado e quando a última linha for executada, o programa será encerrado.*

{

- *Inicia um bloco de código.*

printf(“Alo mundo\n”);

- *Chamada a uma função em C. A função printf é utilizada para imprimir uma mensagem na tela, neste caso, a mensagem Alo mundo que é chamado de parâmetro passado à função.*

}

- *Encerra um bloco de código.*

Blocos de código

- *Por ser uma linguagem estruturada, a linguagem C permite a criação de blocos de código. Um bloco de código é um grupo de comandos de programa conectados logicamente que o computador trata como uma unidade. Para criar um bloco de código, coloque uma seqüência de comandos entre chaves, como pode ser visto no programa exemplo, as linhas 5, 6 e 7 representam um bloco de código.*

Ponto e vírgula

- *O ponto e vírgula é um terminador de comandos, por isso, todos os comandos devem ser terminados por um. Desta forma, podemos ter vários comandos numa mesma linha sendo cada um terminado com um ponto e vírgula.*

Chaves

- *Todo bloco de código escrito em C deve vir entre chaves. Não é necessário colocar um ponto e vírgula depois de fechar chaves pois cada comando dentro do bloco já possui o seu terminador.*

Comentários

- *Na linguagem C, os comentários são delimitados por /* e */ como pode ser visto na primeira linha do programa anterior. Não é permitido colocar comentários aninhados. Os comentários podem vir em qualquer posição do programa e não apenas em linhas separadas. Eles também podem começar em uma linha e terminar em outra.*

Palavras reservadas

- *Como todas as outras linguagens de programação, C consiste em palavras reservadas e em regras de sintaxe que se aplicam a cada palavra reservada. Uma palavra reservada é essencialmente um comando, e na maioria das vezes, as palavras reservadas de uma linguagem definem o que pode ser feito e como será feito. O padrão ANSI C especifica as seguintes palavras reservadas:*

Palavras reservadas

a
b
c

Variáveis, constantes, operadores e expressões

- *Por ser uma linguagem estruturada, em C, as variáveis devem ser declaradas antes de serem usadas, permitindo assim, que o compilador faça checagens em tempo de compilação.*

Identificadores

- *Identificadores são nomes usados para se fazer referência a variáveis, funções, rótulos e vários outros objetos definidos pelo usuário. Um identificador pode ter de um a vários caracteres. O primeiro deve ser uma letra ou um sublinhado, e os caracteres subseqüentes deve serm letras, números ou um sublinhado.*

Tipos de dados

- Em C, existem 5 tipos de dados básicos: caracter, inteiro, ponto flutuante, ponto flutuante de dupla precisão e sem valor. As palavras reservadas para declarar variáveis destes tipos são char, int, float, double e void respectivamente. Veja na tabela a seguir o espaço gasto por cada um destes tipos assim como seus limites em máquinas IBM PC compatíveis.

Tipos de dados

C

Modificadores de tipo

- Com exceção de void, os tipos de dados básicos têm vários modificadores que os precedem. O modificador é usado para alterar o significado do tipo-base para que ele se adapte da maneira mais precisa às necessidades das várias situações. Eis aqui uma lista dos modificadores: signed, unsigned, long, short. Os dois primeiros modificadores indicam a existência ou não de sinal enquanto os outros dois são relativos ao tamanho de memória necessário para armazenar o valor de um elemento deste tipo.

Modificadores de tipo

Declarando variáveis

- *Uma declaração de variável deve seguir a seguinte regra:*

  tipo lista_variáveis;

- *onde tipo deve ser um tipo válido em C e lista_variáveis pode consistir em um ou mais identificadores separados por vírgula.*

Exemplos de declaração de variável

- *int i, j, l;*
- *short int si;*
- *unsigned int ui;*
- *long inteiro_grande;*
- *double balanco, lucro, prejuizo;*

Onde declarar?

- *Existem 3 lugares em um programa C onde as variáveis podem ser declaradas.*

Variável global

- *O primeiro lugar é fora de todas as funções, incluindo a função main(). A variável decalrada dessa maneira é chamada variável global e pode ser usada em qualquer parte do programa.*

Variável local

- *O segundo lugar é dentro de uma função. Estas variáveis são chamadas variáveis locais e podem ser usadas somente pelos comandos que estiverem na mesma função.*

Parâmetro (argumento)

- *O último lugar onde as variáveis podem ser declaradas é na declaração dos parâmetros formais de uma função, embora as variáveis aqui declaradas sejam utilizadas para receber os argumentos quando a função é chamada, eles podem ser utilizados como outra variávei qualquer.*

Programa exemplo:

```
/* soma os números de 0 a 9 */
int soma;                    /* Variável global */
main()
{
      int cont;              /* Variável local */
      soma = 0;              /* inicializa variável soma */
      for (cont = 0; cont < 10; cont ++) {
            total(cont);
            display();
      }
}
```

Função total()

```
/* acumula no total parcial */
total(x)
int x;                          /* Parâmetro formal */
{
      soma = x + soma;
}
```

Função display()

```
display()
{
    int cont;              /* Variável local */
    /* esta variável cont é diferente
      daquela declarada em main() */

    for(cont = 0; cont < 10; cont ++) printf("-");
    printf("A soma atual é %d", soma);
}
```

Comentários

- *Neste exemplo, qualquer função do programa pode acessar a variável global soma. Porém total() não pode acessar diretamente a variável local cont em main(), que deve passar cont como um argumento. Isto é necessário porque uma variável local só pode ser usada pelo código que está na mesma função na qual ela é declarada. Observe que cont em display() é completamente separada de cont em main(), novamente porque uma variável local é conhecida apenas pela função na qual ela é declarada.*

Inicialização de variáveis

- *Nós vimos que em main() existe uma linha somente para inicializar a variável soma, esta linha poderia ser suprimida se a variável fosse declarada*

```
int soma = 0;
```

- *Desta forma, podemos inicializar variáveis no momento de sua declaração, o que facilita muito a escrita do programa além de reduzir o seu tamanho.*

Constantes

- Constantes são valores fixos que o programa não pode alterar.

Exemplos de Constantes

Constantes hexadecimais e octais

- Podem ser declaradas constantes em hexadecimal ou octal conforme o exemplo a seguir:

```
int hex = 0xFF;        /* 255 em decimal */
/* as constantes em hexadecimal devem
   ser precedidas por 0x */
int oct = 011;  /*   9 em decimal */
/* as constantes em octal devem
   ser precedidas por 0 */
```

Constantes strings

- *Uma string é um conjunto de caracteres entre aspas. Por exemplo, “esta é uma string” é uma string. Não confundir strings com caracteres, ‘a’ é um caracter enquanto “a” é uma string.*

Constantes com barras invertidas

- Existem alguns caracteres que não podem ser representados no texto comum, as constantes com barra invertida servem para representar estes caracteres.

Constantes com barras invertidas

Operadores

- *A linguagem C é muito rica em operadores. Os operadores são divididos em 3 categorias gerais: aritméticos, de relação e lógicos e bit a bit. Além desses, C tem operadores especiais para tarefas particulares.*

Operadores aritméticos

Exemplo:

```
main()
{
    int x = 10, y = 3;
    printf("%d\n", x / y);        /* exibirá 3 */
    printf("%d\n", x % y);
     /* exibirá 1, o resto da divisão de inteiros */
    x = 1;
    y = 2;
    printf("%d %d\n", x / y, x % y);
    /* exibirá 0 e 1 */
```

Exemplo (continuação)

```
    x ++;
    printf("%d\n", x);  /* exibirá 2*/
    printf("%d %d\n", x++, ++y);/* exibirá 2 e 3 */
    /* neste caso, x só é incrementado depois que o
    comando
       é executado enquando y é incrementado antes */
}
```

Operadores de relação

Operadores lógicos

Observação

- *Em C, o número 0 representa falso e qualquer número não-zero é verdadeiro, assim, os operadores anteriores retornam 1 para verdadeiro e 0 para falso.*

Operador de atribuição

- O operador = é o operador de atribuição. Ao contrário de outras linguagens, C permite que o operador de atribuição seja usado em expressões com outros operadores.

```
int a, b, c;
a = b = c = 1;  /* atribui 1 às 3 variáveis */
((a = 2 * b) > c)        /* a = 2 e a comparação
   resulta em 1 */
```

Expressões

- *Os operadores, as constantes e as variáveis são os componentes das expressões. Uma expressão em C é qualquer combinação válida desses componentes.*

Conversões de tipos

- Quando você mistura constantes e variáveis de tipos diferentes em uma expressão, C as converte para o mesmo tipo. O compilador C converterá todos os operandos para o tipo do maior operando, uma operação de cada vez, conforme descrito nestas regras de conversão de tipo:

Regra 1

- Todo char e short int é convertido para int. Todo float é convertido para double.

Regra 2

- *Para todos os pares de operandos, ocorre o seguinte, em seqüência: se um dos operandos for um long double, o outro será convertido para long double. Se um dos operandos for double, o outro será convertido para double. Se um dos operandos for long, o outro será convertido para long. Se um dos operandos for unsigned, o outro será convertido para unsigned.*

Type cast

- *Pode-se forçar o compilador a efetuar determinada conversão utilizando-se um type cast que tem a seguinte forma:*

  (tipo) expressão

Exemplo:

```
main()
{
    int x = 3;
    printf("%f %f\n", (float) x / 2, (float) (x / 2));
/* serão impressos na tela 1.5 e 1.0 pois no primeiro
    caso, x é convertido para float e depois é dividido, já
    no segundo, somente o resultado é convertido para
    float */
}
```

Modificadores de acesso

- *Os modificadores de acesso informam sobre como será feito o acesso à variável.*

Register

- *Sempre que uma variável for declarada do tipo register, o compilador fará o máximo possível para mante-la num dos registradores do microprocessador, acelerando assim o acesso a seu valor. É pratica comum, declarar as variáveis de controle de loop como sendo register.*

Static

- *Variáveis static são variáveis existem durante toda a execução do programa, mas só podem ser acessadas de dentro do bloco que a declarou.*

Exemplo:

```
/* Exemplo de variável static */
main()
{
   printf("%d\n", numero());      /* imprimirá 0 */
   printf("%d\n", numero());      /* imprimirá 1 */
}
numero()
{
   static valor = 0; /* atribuição só executada 1 vez */
   return valor ++;
}
```

printf()

- *Rotina de finalidade geral para saída pelo console*
- *A função printf() serve para mostrar mensagens na tela. Sua forma geral é*

printf(“string de controle”, lista argumentos);

“string de controle”

- *A string de controle consiste em dois tipos de itens:*
  - *Caracteres que a função imprimirá na tela*
  - *Comandos de formatação*

Comandos de formatação

- Todos os comandos de formatação começam por % que é seguido pelo código de formatação
- Deve haver exatamente o mesmo número de argumentos quanto forem os comandos de formatação e eles devem coincidir em ordem

Comandos de formatação

Comprimento mínimo do campo

- *Para especificar o comprimento mínimo que um campo poderá ter, basta colocar um inteiro entre o sinal % e o comando de formatação. Observe que este é o comprimento mínimo e que o campo pode ocupar um espaço maior.*

Número de casas decimais

- Para especificar o número de casas decimais, coloque um ponto decimal após o especificador de largura mínima do campo e depois dele, o número de casas decimais que deverão ser exibidas.

Comprimento máximo

- *Quando o formato de casas decimais é colocado em strings, o número de casas decimais passa a ser considerado como comprimento máximo do campo.*

Número mínimo de dígitos

- *Quando o formato de casas decimais é aplicado em inteiros, o especificador de casas decimais será utilizado como número mínimo de dígitos*

Alinhamento

- Por padrão, todo resultado é alinhado à direita, para inverter este padrão, utilize um sinal de menos (-) antes de especificar o tamanho.

Modificadores de formatação

- Podemos usar modificadores para informar sobre a leitura de shorts (modificador h) ou longs (modificador l)

Exemplos

scanf()

- *Rotina de finalidade geral para entrada pelo console*
- *A função scanf() serve para ler informações do teclado. Sua forma geral é*

scanf(“string de controle”, lista argumentos);

“string de controle”

- *A string de controle consiste em três classificações de caracteres:*
  - *Especificadores de formato*
  - *Caracteres brancos*
  - *Caracteres não-brancos*

Especificadores de formato

- Todos os especificadores de formato começam por % que é seguido por um caracter que indica o tipo de de dado que será lido
- Deve haver exatamente o mesmo número de argumentos quanto forem os especificadores de formato e eles devem coincidir em ordem

Especificadores de formato

Caracter branco

- *Um caracter branco na string de controle faz com que scanf() passe por cima de um ou mais caracteres brancos na string de entrada*
- *Um caracter branco é um espaço, um tab ou um \n*

Caracter não branco

- *Um caracter não branco na string de controle faz com que scanf() leia e desconsidere um caracter coincidente. Se o computador não encontrar o caracter especificado, scanf terminará*
- *O comando "%d,%d" fará com que scanf leia um inteiro, depois leia e desconsidere uma vírgula e finalmente, leia um outro inteiro*

Como chamar scanf()

- *Todas as variáveis usadas para receber valores através de scanf() devem ser passadas por seus endereços. Se quiser ler a variável cont, utilize*

  scanf(“%d”, &cont);

- *Como strings são representadas por vetores, NÃO deve ser colocado o & antes do nome da variável*

Ignorar entrada

- Colocar um * entre o % e o código de formatação fará com que scanf() leia dados do tipo especificado mas suprimirá suas atribuições. Desta forma

  scanf("%d%*c%d", &x, &y)

- dada a entrada 10/20, coloca 10 em x, desconsidera o sinal de divisão e coloca 20 em y

Comprimento máximo do campo

- Para especificar o comprimento máximo que um campo poderá ter, basta colocar um inteiro entre o sinal % e o comando de formatação. Os caracteres que sobrarem serão utilizados nas próximas chamadas a scanf(). Caso não queira ler mais do que 20 caracteres na string nome, utilize

  scanf(“%20s”, nome);

Espaços, tabs e \n

- *Servem como separadores quando não estiverem sendo lidos caracteres. São lidos e atribuídos quando for pedido um caracter de entrada. Caso o comando*

  scanf(“%c%c%c”, &a, &b, &c);

- *seja lido com a entrada*

  x y

- *scanf retornará com x em a, espaço em b e y em c*

Comandos de controle de fluxo

- *Os comandos de controle de fluxo são a base de qualquer linguagem.*
- *A maneira como eles são implementados afeta a personalidade e percepção da linguagem.*
- *C tem um conjunto muito rico e poderoso de comandos de controle de fluxo.*
- *Eles se dividem em comandos de teste de condições e comandos de controle de loop.*

Comandos de testes de condições

- *Estes comandos avaliam uma condição e executam um bloco de código de acordo com o resultado. São eles:*
  - *if*
  - *switch*

if

- *O comando if serve para executar comandos de acordo com uma determinada condição*
- *A forma geral do comando if é*

  if (condição) comando;

  else comando;
- *onde a parte else é opcional*

Exemplo

```
/* programa do número mágico */
#include <stdlib.h>
main()
{
   int magico, adivinhacao;
   magico = rand() % 10;  /* gerar um número */
   printf("Adivinhe o número: ");
   scanf("%d", &adivinhacao);
   if (adivinhacao == magico)
     printf("** número certo **");
   else
     printf("-- número errado --");
}
```

if aninhados

- *C permite que sejam colocados comandos if dentro de outros comandos if. A isto chamamos de if aninhados.*
- *Quando se trata de if aninhados, o comando else se refere ao if mais próximo que não possui um comando else. Tanto o if quanto o else devem estar dentro do mesmo bloco de código.*

Exemplos

```
if (x)
    if (y) printf("1");
    else printf("2");
```

- Neste caso, o else pertence ao segundo if.

```
if (x) {
    if (y) printf("1");
}
else printf("2");
```

- Neste caso, o else pertence ao primeiro if.

if-else-if

- *É muito comum encontrar programas que possuem uma "escada" if-else-if da seguinte forma:*

```
if (condição)
   comando;
else if (condição)
   comando;
else
   comando;
```

Avaliação do if-else-if

- *O computador avalia as expressões condicionais de cima para baixo. Assim que encontra uma condição verdadeira, ele executa o comando associado a ela e passa por cima do resto da "escada".*
- *Se nenhuma condição for verdadeira, o computador executará o else final.*

A expressão condicional

- Qualquer expressão válida em C pode servir como expressão condicional. Veja o exemplo:

```
/* dividir o primeiro número pelo segundo */
main()
{
    int a, b;
    printf("Digite dois números: ");
    scanf("%d %d", &a, &b);
    if (b) printf("%d\n", a/b);
    else printf("não posso diridir por zero\n");
}
```

switch

- *Embora o if-else-if possa executar vários tipos de testes, o código pode ficar muito difícil de ser seguido.*
- *C possui um comando de vários desvios chamado switch.*
- *No switch, o computador testa uma variável sucessivamente contra uma lista de constantes inteiras ou de caracteres e executa um comando ou bloco de comandos quando encontrar uma coincidência.*

Forma geral do switch

```
switch (variável) {
   case constante1:
     seqüência de comandos
     break;
   case constante2:
     seqüência de comandos
     break;

   default:
     seqüência de comandos
}
```

O comando default dentro do switch

- *O comando default será executado se não for encontrada nenhuma coincidência na lista de constantes.*
- *Caso não seja colocado um comando default e não haja coincidência, nenhum comando será executado.*

O comando break

- Quando o computador encontra alguma coincidência, ele executa os comandos associados àquele case até encontrar break ou o fim do comando switch.
- É um erro comum programadores esquecerem de colocar o break após os comandos.

Importante

- *O switch difere do if, já que o primeiro só pode testar igualdade e a expressão condicional if pode ser de qualquer tipo.*
- *Não pode haver duas constantes case com valores iguais no mesmo switch.*
- *Podem ser colocados comandos switch dentro de comandos switch.*
- *Pode ser deixado um case vazio quando mais de uma condição usa o mesmo código.*

Comandos de controle de loops

- *Os comandos de controle de loops permitem que o computador repita um conjunto de instruções até que alcance uma certa condição. Em C temos os seguintes comandos de controle de loop:*
  - *for*
  - *while*
  - *do while*

for

- *O loop for em C é muito mais forte e mais flexível que o da maioria das outras linguagens. Sua forma geral é*

  for (inicialização; condição; incremento)

  comando;

- *Observe que as três partes do loop for são separadas por ponto e vírgula.*
- *Nenhuma destas partes precisa existir.*

Inicialização

- *Na forma mais simples, inicialização é um comando de atribuição que o compilador usa para estabelecer a variável de controle de loop.*
- *A inicialização pode conter qualquer comando válido em C.*

Condição

- A condição é uma expressão de relação que testa se a condição final desejada pelo loop for ocorreu.
- Aqui também pode ser colocado qualquer comando válido em C.

Incremento

- *O incremento define a maneira como a variável de controle do loop será alterada cada vez que o computador repetir o loop.*
- *Também aqui, podemos colocar qualquer comando válido em C.*

Exemplo 1

```
/* imprimir os números de 1 a 100 */
main()
{
   int x;
   for (x = 1; x <= 100; x ++) printf("%d ", x);
}
```

Exemplo 2

```
/* imprimir os números de 100 a 1 */
main()
{
   int x;
   for (x = 100; x > 0; x --) printf("%d ", x);
}
```

Exemplo 3

```
/* imprimir os números de 0 a 100, 5 em 5 */
main()
{
   int x;
   for (x = 0; x <= 100; x = x + 5)
     printf("%d ", x);
}
```

Exemplo 4

```
/* executa um bloco de código 100 vezes */
main()
{
   int x;
   for (x = 0; x < 100; x ++) {
     printf("O valor de x é: %d ", x);
     printf("e o quadrado de x é: %d\n", x * x);
   }
}
```

Variações do loop for

- Podem ser executados mais de um comando nas partes de inicialização e de incremento. Veja que

```
main()
{
   int x, y;
   for (x = 0, y = 0; x + y < 100; ++x, y++)
     printf("%d ", x + y);
}
```

- Mostrará números de 0 a 98, 2 a 2.

Um uso diferente para for

```
main()
{   int t;
    for (prompt(); t=readnum(); prompt()) sqrnum(t);
}
prompt()
{   printf("digite um inteiro:" );
}
readnum()
{   int t;
    scanf("%d", &t);
    return t;
}
sqrnum(int num)
{   printf("%d\n", num * num);
}
```

Loop infinito

- *Podemos fazer um comando for executar para sempre simplesmente não especificando sua parte condicional. Veja*

```
for (;;)
    printf("este loop rodará para sempre\n");
```

Saindo de um loop

- Podemos usar o comando break para encerrar um for a qualquer momento. Veja um exemplo:

```
main()
{
   int a;
   for (a = 1; a < 100; a++)
     if (a == 10) break;
}
```

- O loop só será executado 10 vezes.

Loops for sem nenhum corpo

- Podem ser utilizados loops sem corpo para gerar retardo de tempo. Veja um exemplo:

```
for (a = 0; a < 1000; a ++);
```

while

- *O while executa um comando (ou bloco de comandos) enquanto uma condição for verdadeira.*
- *A forma geral do while é:*

```
while (condição)
    comando;
```

Exemplo 1

```
pausa()
{
   char tecla = '\0';
   printf("Tecle ESPAÇO para continuar...");
   while (tecla != ' ')
     tecla = getche();
}
```

Exemplo 2

- O exemplo anterior pode ser reescrito sem utilizar variável. Veja:

```
main()
{
   printf("Tecle ESPAÇO para continuar...");
   while (getche() != ' ');
}
```

do while

- Ao contrário do loop for e do loop while, que testam a condição no começo do loop, o loop do while verifica a condição somente no final. Desta forma, o loop será executado pelo menos uma vez. A forma geral do loop do while é:

```
do {
   comando;
} while (condição);
```

Exemplo

```
/* Lê um número maior que 100 */
main()
{
   int num;
   do {
     printf("Digite um número maior que 100");
     scanf("%d", &num);
   } while (num <= 100);
}
```

Loops aninhados

- *Quando um loop está dentro do outro, dizemos que o loop interior está aninhado.*
- *Loops aninhados permite que sejam solucionados vários problemas de programação.*
- *Em C podemos aninhar qualquer tipo de loop.*

Exemplo

```
/* exibir uma tabela de potências dos números de 1 a 9*/
main()
{   int i, j, k, temp;
    printf("\ti\ti^2\ti^3\ti^4\n");
    for (i = 1; i < 10; i ++) {      /* loop externo */
      for (j = 1; j < 5; j ++) {     /* primeiro nível de aninhamento */
                temp = 1;
                for (k = 0; k < j; k ++)      /* loop mais interior */
                          temp = temp * i;
                printf("%8d", temp);
      }
    }
}
```

Interrupção de loops

- Quando precisamos encerrar um loop sem que sua condição de encerramento esteja satisfeita, podemos utilizar o comando break.
- O comando break é especialmente útil quando algum evento externo controle um loop.

Comando continue

- O comando continue funciona de maneira parecida com o comando break. Porém, em vez de forçar o encerramento, continue força a próxima iteração do loop e pula o código que estiver no meio.

Exemplo

```
/* programa para imprimir os números pares
   entre 0 e 98 */
main()
{
   int x;
   for (x = 0; x < 100; x ++) {
     if (x % 2) continue;
     printf("%d ", x);
   }
}
```

Vetores

- *Vetores são uma coleção de variáveis do mesmo tipo que são referenciadas pelo mesmo nome.*
- *Em C, um vetor consiste em locações contíguas de memória.*
- *O elemento mais baixo corresponde ao primeiro elemento e o mais alto ao último.*
- *O vetor mais utilizado é o de caracteres.*

Declarando vetores

- A forma geral da declaração de um vetor é:

  tipo nome_var[tamanho];

- Onde
  - tipo é o tipo base do vetor e
  - tamanho é a quantidade de elementos que o vetor conterá

Acessando um vetor

- Os vetores são acessados através de índices colocados entre colchetes.
- O índice do primeiro elemento do vetor é 0 (ZERO).

```
int amostra[10];      /* vetor de 10 inteiros */
amostra[0]            /* primeiro elemento */
amostra[9]            /* último elemento */
```

Exemplo

```
main()
{
   int x[10]; /* vetor com 10 elementos int */
   int t;

   for (t = 0; t < 10; t ++)
            x [ t ] = t;
}
```

Limites dos vetores

- C não faz checagem dos limites dos vetores, isto é responsabilidade do programador. Logo, o código a seguir não causará nenhum erro.

```
int elementos[10];
elementos[12] = 0;
elementos[10] = 0;
```

Strings

- *Uma string é por definição, um vetor de caracteres terminado em 0. Então, para declarar a string, devemos declarar sempre um elemento a mais para o terminador. Veja que*

  char mensagem[10] = “Exemplo”

- *Ficará armazenado na memória como:*

Funções para manipular strings

gets

- *Lê uma string do teclado.*
- *Forma geral:*

  gets(nome_string);
- *Exemplo:*

```
main()
{  char str[80];
   gets(str);
   printf("%s", str);
}
```

puts

- Mostra uma string na tela.
- Forma geral:

```
puts(nome_string);
```

- Exemplo:

```
main()
{
   puts("Esta é uma mensagem");
}
```

strcpy

- Copia uma string em outra.
- Forma geral:

```
strcpy(para, de);
```

- Lembre-se que a string para deve ser grande o suficiente para conter de.

```
main()
{  char str[80];
   strcpy(str, “alo”);
}
```

strcat

- Adiciona uma string em outra.
- Forma geral:

  strcat(s1, s2);

- s2 será anexada ao final de s1.

```
main()
{   char primeiro[20], segundo[10];
    strcpy(primeiro, “bom”);
    strcpy(segundo, “ dia”);
    strcat(primeiro, segundo);
    printf(“%s\n”, primeiro);
}
```

strcmp

- *Compara 2 strings.*
- *Forma geral:*

  strcmp(s1, s2)

- *A função retorna:*

strlen

- Retorna o tamanho da string
- Forma geral:

  strlen(str);
- Exemplo:

```
main()
{  char str[80];
   printf("Digite uma string: ");
   gets(str);
   printf("Tamanho: %d\n", strlen(str));
}
```

Matrizes bidimensionais

- C permite que sejam declaradas matrizes bidimensionais.
- Forma da declaração:

  tipo nome_var[dimensão1][dimensão2];

- Exemplo:

  char tabela[5][5];

Matrizes multidimensionais

- De forma semelhante as matrizes bidimensionais, declaramos as multidimensionais. Veja por exemplo uma matriz de 4 dimensões:

```
int matriz[5][7][3][8];
```

Acessando os elementos das matrizes multidimensionais

```
main()
{
   int numeros[4][3], i, j;
   for (i = 0; i < 4; i ++)
     for (j = 0; j < 3; j++)
              numeros[ i ][ j ] = i * j;
}
```

Matrizes de strings

- Uma matriz bidimensional de caracteres pode ser interpretada como uma matriz de strings.

```
main()
{  char strings[5][20]; /* 5 strings */
   int i;
   for (i = 0; i < 5; i ++)
     gets(strings[ i ]);
}
```

Inicialização de matrizes

- C permite que as matrizes globais sejam inicializadas.
- A forma geral é:

  tipo nome_matriz[tam1]...[tamN] = {lista de valores}

- Exemplo:

  int i[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

Exemplo

```
int quadrados[5][2] = {
    1, 1,
    2, 4,
    3, 9,
    4, 16,
    5, 25};
```

quadrados =

Declarando vetores sem tamanho

- Podem ser declarados vetores sem especificar explicitamente seus tamanhos. Os vetores devem ser inicializados na declaração. O tamanho será definido na inicialização.
- Exemplo:

```
char mensagem[ ] = “Esta é uma string”;
```

Exercícios

- *Faça um programa que leia 3 nomes com suas respectivas idades e mostre o maior e o menor.*
- *Faça um programa que leia 5 números e imprima "Números altos" se mais do que 3 deles forem maiores que 5, caso contrário, imprima "Números baixos".*

Modelos de memória

- A definição de modelos de memória foi a forma utilizada para contornar as limitações de segmentos em computadores IBM PC.

Ponteiros

- *Entender e usar corretamente os ponteiros são pontos cruciais para criação de programas bem-sucedidos em C.*
- *Além dos ponteiros serem uma das características mais fortes em C, também é a mais perigosa.*
- *É muito fácil usar ponteiros incorretamente causando erros difíceis de serem encontrados.*

Motivos para usar ponteiros

- *Os ponteiros fornecem maneiras pelas quais as funções podem modificar os argumentos recebidos.*
- *Dão apoio às rotinas de alocação dinâmica.*
- *Podem substituir o uso vetores em muitas situações para aumentar a eficiência.*

Ponteiros são endereços

- *Um ponteiro é uma variável que contém um endereço de memória. Isto é, eles possuem armazenam a localização de outra variável dentro da memória do computador.*
- *Então dizemos que um ponteiro aponta para esta variável.*

Ilustração de poteiros

Variáveis ponteiros - declaração

- A declaração de variáveis ponteiros, segue a seguinte regra geral:

  tipo *nome_var;

- onde tipo é o tipo do elemento para o qual o ponteiro apontará.
- Exemplo:

  char *p;

  int *temp, *valor;

Os operadores de ponteiros

- Existem 2 operadores especiais de ponteiros: & e *.
- O operador & devolve o endereço da variável. É utilizado para fazer um ponteiro apontar para ela.
- O operador * devolve o valor armazenado no endereço apontado pelo ponteiro.

Exemplo

```
main()
{  int numero = 5, *p;
   p = &numero;
   printf("%d %d\n", numero, *p);
   *p = 3;
   printf("%d %d\n", numero, *p);
   numero = 7;
   printf("%d %d\n", numero, *p);
}
```

Expressões com ponteiros

- C permite que sejam feitas expressões com ponteiros e elas seguem as mesmas regras das outras expressões em C.
- Quando se compara um ponteiro com outro, estamos comparando seus endereços. Isto é útil quando ambos os ponteiros apontam para elementos de um vetor.

Ponteiros e vetores

- Existe um relacionamento muito próximo entre os ponteiros e os vetores. Veja o código:

```
char str[80], *p;

p = str;
```

- Este código faz com que p aponte para o primero elemento do vetor, pois um vetor sem o índice se comporta como um ponteiro para seu primeiro elemento.

Exemplo

- Após a definição:

  char str[80], *p;

  p = str;

- são equivalentes os acessos ao quinto elemento de str:

  str[4]

  *(p + 4)

  p[4]

Problemas com ponteiros

- *É muito fácil errar quando se trabalha com ponteiros em C.*
- *Algumas vezes, os erros com ponteiros só aparecem quando o programa cresce.*
- *Ponteiros que não foram inicializados, apontam para um lugar desconhecido na memória, que pode ser inclusive o código do programa.*

Dois erros comuns

```
/* este programa está
   errado */

main()
{
   int x, *p;
   x = 10;
   *p = x;
}
```

```
/* este programa está
   errado */

main()
{
   int x, *p;
   x = 10;
   p = x;
   printf("%d", *p);
}
```

Alocação dinâmica de memória

- *A linguagem C permite que seja feita alocação dinâmica de memória, isto é, solicitar memória a medida que ela for sendo necessária.*
- *Programas que utilizam listas, pilhas, e outros tipos de dados complexos cuja quantidade máxima não é definida durante a compilação, trabalham com alocação dinâmica de memória.*

Funções de alocação dinâmica

- Alocar um bloco de memória

  void *malloc(int número_de_bytes);

  void *calloc(unsigned num, unsigned tam);

- Liberar um bloco de memória

  void free(void *p);

- Alterar o tamanho de um bloco de memória

  void *realloc(void *p, unsigned tam);

Exemplo 1

```
#include <stdio.h>
#include <stdlib.h>
main()
{  char *str;
   str = (char *) malloc(101 * sizeof(char));
   if (str) {
     strcpy(str, “Isto é um teste”);
     printf(“%s”, str);
     free(str);
   }
}
```

Exemplo 2

```
#include <stdio.h>
#include <stdlib.h>
main()
{  float *p;
   p = (float *) calloc(100, sizeof(float));
   if (!p) {
     printf("alocação fracassada - abortado");
     exit(1);
   }
   free(p);
}
```

Funções

- *Funções são os blocos de contrução em que ocorrem todas as atividades do programas.*
- *Programar utilizando funções é reduzir a complexidade do código e simplificar sua escrita.*

Forma geral

- A forma geral da declaração de uma função é:

```
tipo nome_função(lista_parâmetros)
declaração de parâmetros
{
   corpo da função
}
```

- Caso não seja especificado o tipo da função, ela será do tipo int.

Comando return

- O comando return possui duas utilidades básicas:
  - Causar a saída imediata da função na qual ele se encontra, retornando para o código de chamada.
  - Devolver um valor para a função chamadora.

Exemplo

```
/* retorna 1 se o parâmetro > 10 */
int maior_que_dez(int x)
{
   return (x > 10);
}
```

Argumentos

- *Quando é necessário passar alguma informação extra para uma função, esta informação será passada através de argumentos.*
- *Para chamar uma função com argumentos, eles devem ser colocados entre parênteses após o identificador da função. Veja:*

  puts(“Atenção”);

- *a string “Atenção” é o argumento passado para a função puts.*

Chamadas por valor e por referência

- *Existem duas formas de se passar um parâmetro para uma função, utilizando chamadas por valor ou por referência.*
- *Quando um parâmetro é passado por valor, as alterações efetuadas na função não afetarão a variável da função chamadora.*
- *Quando um parâmetro é passado por referência, qualquer alteração efetuada pela função afetará a variável da função chamadora.*

Exemplo

```
main()
{  int x = 3, y = 4;
   printf("%d %d\n", x, y);
   processa(x, &y);
   printf("%d %d\n", x, y);
}

int processa(int a, int *b)
{  a = 6;
   b = 7;
}
```

Observações

- *Vetores são passados por referência.*
- *Quando uma função é feita para receber um parâmetro por referência, o parâmetro deve ser declarado como um ponteiro na lista de argumentos da função.*
- *Para conjuntos grandes de dados, a passagem de parâmetros por referência é mais rápida.*

Argumentos argc e argv para main()

- *A função main também recebe argumentos. Eles são os parâmetros da linha de comando passados ao programa. Os dois parâmetros são argc e argv. Veja como eles devem ser declarados:*

  ```
  main(int argc, char *argv[ ])
  ```

- *Argc conterá o número de parâmetros passados e argv conterá os parâmetros.*

Exemplo

```
/* Função para imprimir os parâmetros */
main(int argc, char *argv[ ])
{
   int i;
   for (i = 0; i < argc; i ++)
     printf("argv[%d]: %s\n", i, argv[i]);
}
```

Funções que devolvem não inteiros

- *Quando o tipo de uma função não é declarado, por default, ele será int.*
- *Se for necessário definir uma função que retorna valores de outros tipos, o tipo deve ser especificado.*
- *Quando a função não é do tipo int, ela deve ser identificada antes de ser chamada da primeira vez. Assim, o compilador poderá gerar um código correto para a chamada à função.*

Protótipos

- Como vimos, as funções que não retornam int, devem ser identificadas antes de serem referenciadas, para isto, deve ser definido um protótipo para a função.
- Forma geral

  tipo nome_função(tipo_parâmetros);

Exemplo

```
float metade(float);
main()
{
   printf("%f\n", metade(3));
}
float metade(float numero)
{
   return (numero / 2);
}
```

Recursividade

- *A linguagem C permite que as próprias funções se chamem. A esta característica damos o nome de recursividade.*
- *Existem muitos problemas que se tornariam extremamente difíceis de serem implementados sem a recursividade.*

Preprocessador

- *Antes do programa ser compilado, ele é submetido ao preprocessador. Esta característica é muito útil.*
- *Todos os comandos do preprocessador são iniciados por um sinal #, sendo os dois mais usados:*
  - *#define*
  - *#include*

#define

- *O comando define serve para definir um identificador e uma string. O compilador substituirá o identificador pela string toda vez que for encontrado no arquivo-fonte. O identificador é chamado de nome de macro, e o processo de substituição é chamado de substituição de macro.*

  #define indentificador string

Exemplo

```
#define mensagem “Isto é um teste.\n”
#define verdadeiro 1
#define falso !verdadeiro

main()
{
   if (verdadeiro) printf(mensagem);
   if (falso) printf(mensagem);
}
```

#include

- *Instrui o compilador a incluir um outro arquivo fonte com aquele que contém o comando #include. O nome do arquivo a ser incluído deve estar entre aspas ou entre o sinal de maior e menor.*
- *Se o nome do arquivo for colocado entre aspas, o compilador procurará pelo arquivo na seguinte seqüência: diretório atual, diretórios configurados no compilador e diretórios padrões. Caso o nome do arquivo esteja entre < >, não será procurado no diretório atual.*

Entrada e saída

- *A entrada e saída em C é feita utilizando-se funções da biblioteca do compilador, não existe nenhuma palavra reservada para esta finalidade.*
- *Existem dois sistemas de entrada e saída em C:*
  - *Bufferizado*
  - *Não bufferizado*
- *O padrão ANSI só define o primeiro.*

Arquivo cabeçalho stdio.h

- *Muitas bibliotecas em C exigem que certos tipos de dados ou outras informações sejam partes dos programas que as usam. Para isto, é necessário utilizar o comando #include e incluir arquivos cabeçalho com estas definições nos programas.*

  #include <stdio.h>

Filas de bytes

- *O sistema de arquivo bufferizado destina-se a trabalhar com uma variedade de dispositivos, que inclui terminais, acionadores de disco e acionadores de fita.*
- *Embora cada dispositivo seja diferente, o sistema de arquivo bufferizado transforma cada um em um dispositivo lógico chamado fila de bytes.*

Arquivos

- *Em C, um arquivo é um conceito lógico que o sistema pode aplicar a qualquer coisa, desde arquivos em disco até terminais.*
- *Para associar uma fila de bytes a um determinado arquivo, basta realizar uma operação de abertura.*
- *Todas as filas de bytes são iguais.*
- *Nem todos os arquivos são iguais.*

Sistema de entrada e saída buferizado

- *O sistema de entrada e saída bufferizado é composto de várias funções relacionadas.*
- *Para utiliza-las, é necessário incluir o arquivo-cabeçalho stdio.h no programa.*

O ponteiro do arquivo

- *É o que mantém unido o sistema de entrada e saída bufferizado.*
- *É um ponteiro para a informação que define vários aspectos do arquivo, incluindo nome, status e posição corrente.*
- *Um ponteiro de arquivo é uma variável de ponteiro do tipo FILE, que é definida em stdio.h.*

Algumas funções

fopen()

- *Abre uma fila de bytes para ser usada e liga um arquivo a esta fila.*
- *Protótipo:*

  *FILE *fopen(char *nome_arq, char *modo);*
- *onde modo é uma string que contém o status de abertura desejado.*
- *nome_arq é uma string com um nome de arquivo válido para o sistema operacional, podendo incluir drive e diretório.*
- *Retorna NULL (= 0) se a função falhar.*

Valores legais de modo

fclose()

- *Fecha uma fila que foi aberta com fopen().*
- *Todas as filas devem ser fechadas antes que o programa termine.*
- *Protótipo:*

```
int fclose(FILE *arquivo);
```

ferror() e rewind()

- *ferror() determina se a última operação com arquivos produziu um erro (devolvendo 1 caso tenha ocorrido).*
- *rewind() reposiciona no início do arquivo.*
- *Protótipos:*

```
int ferror(FILE *arquivo);
void rewind(FILE *arquivo);
```

getc() e putc()

- *São utilizados para ler e gravar caracteres numa fila previamente aberta.*
- *Protótipo:*

  *int putc(int caracter, FILE *arquivo);*

  *int getc(FILE *arquivo);*
- *putc devolve o caracter gravado em caso de sucesso, em caso de erro EOF(definido em stdio.h) será devolvido.*
- *getc devolve um inteiro mas com o byte superior zero. Ele devolve EOF quando alcançar o fim do arquivo.*

fprintf() e fscanf()

- *São os correspondentes a printf e scanf do sistema de entrada e saída do console. A diferença é que o primeiro parâmetro é um ponteiro de arquivo.*
- *Protótipos:*

```
int fprintf(FILE *arq, const char *controle, ...);
int fscanf(FILE *arq, const char *controle, ...);
```

fgets() e fputs()

- *São as correspondentes a gets e puts para arquivos.*
- *Protótipos:*

  *char *fputs(char *str, FILE *arquivo);*

  *char *fgets(char *str, int comprimento, FILE *arquivo);*

- *Observe que em fgets pode ser especificado o comprimento máximo da string e, ao contrário do gets, o \n final é colocado na string.*

getw() e putw()

- *São utilizadas para ler e gravar inteiros.*
- *Elas trabalham exatamente como getc e putc com exceção de gravarem inteiros.*
- *Protótipos:*

```
int getw(FILE *arquivo);
int putw(int  numero, FILE *arquivo);
```

fread() e fwrite()

- *São funções utilizadas para ler e gravar blocos de dados nos arquivos.*
- *Protótipos:*

  *int fread(void *buffer, int tamanho, int quantidade, FILE *arquivo);*

  *int fwrite(void *buffer, int tamanho, int quantidade, FILE *arquivo);*

- *buffer é um ponteiro para a região da memória que possui os dados, tamanho é o tamanho em bytes de cada unidade, quantidade determina quantos itens (cada um contendo tamanho bytes) serão lidos ou gravados e arquivo é um ponteiro de arquivo para uma fila previamente aberta.*

fseek()

- *É utilizado para efetuar operações de leitura e gravação aleatórias sob o sistema de entrada e saída bufferizado.*
- *Protótipo:*

  *int fseek(FILE *arquivo, long int num_bytes, int origem);*
- *fseek retorna zero em caso de sucesso.*
- *Arquivo é o ponteiro de arquivo, num_bytes é o número de bytes desde origem até chegar a nova posição e origem é um dos seguintes macros:*

As filas de bytes stdin, stdout e stderr

- *Existem 3 filas de bytes especiais que são abertas automaticamente quando o programa é inicializado:*
  - *stdin ➡ entrada padrão*
  - *sdtout ➡ saída padrão*
  - *stderr ➡ erro padrão*
- *Elas podem ser usadas para efetuar operações de entrada e saída no console.*

Exemplo

```
#include <stdio.h>
main(int argc, char *argv[ ])
{  FILE *entrada, *saida;
   char ch;
   if (argc != 3) {
     printf("Não foi digitado o nome da origem.\n");
     exit(1);
   }
   if (!(entrada = fopen(argv[1], "rb")) {
     printf("arquivo de origem não achado.\n");
     exit(1);
   }
```

Exemplo (continuação)

```
    if (!(saida = fopen(argv[2], “wb”)) {
      printf(“Arquivo destino não pode ser aberto.\n”);
      exit(1);
    }
                /* esta é a linha que copia o arquivo */
    while (!feof(entrada)) putc(getc(entrada), saida);
    fclose(entrada);
    fclose(saida);
}
```

Tipos de dados definidos pelos usuários

- *Em C podem ser criados 5 tipos diferentes de dados personalizados:*
  - *estrutura*
  - *campo de bit*
  - *união*
  - *enumeração*
  - *typedef*
- *O uso de tipos definido pelo usuário facilita a programação e dá maior poder ao programador.*

Estruturas

- *Em C, uma estrutura é uma coleção de variáveis que são referenciadas pelo mesmo nome.*
- *É uma forma conveniente de manter juntas informações relacionadas.*
- *Forma geral:*

```
struct nome_estrutura {
    tipo1 var1;
    tipo2 var2;
} var_estrutura;
```

Declarando variáveis do tipo estrutura

- Além de poder declarar variáveis do tipo da estrutura durante a definição da estrutura, elas podem ser declaradas da seguinte forma:

  struct nome_estrutura nome_variável;

Acessando variáveis do tipo estrutura

- *Para acessar variáveis do tipo estrutura, utiliza-se o operador . (ponto).*
- *Forma geral:*

  nome_variavel.nome_elemento;

Exemplo

```
struct pessoa {
   char nome[21];
   int idade;
} primeiro;
main() {
   struct pessoa segundo;
   strcpy(primeiro.nome, “José”);
   primeiro.idade = 20;
   segundo = primeiro;
}
```

Vetores e matrizes de estruturas

- Podem ser declarados vetores e matrizes de estruturas, para isto, usamos a forma geral:

  struct nome_estrutura nome_var[t1][t2]...[tn];

- Exemplo:

  struct pessoa pessoas[5];

Ponteiros para estruturas

- Em C, podem ser declarados ponteiros para estruturas.
- Forma geral:

```
struct pessoa *primeiro;
```

Vantagens de se usar ponteiros para estruturas

- *Fazer chamada por referência para uma função;*
- *Criar listas ligadas com estruturas dinâmicas de dados usando o sistema de alocação.*
- *É mais rápido passar estruturas grandes por referência (usando ponteiros) do que por valor, pois estamos passando apenas um endereço.*

Acessando os elementos usando ponteiros para estruturas

- Veja a declaração:

```
struct pessoa *primeiro, segundo;
strcpy(segundo.nome, “José”);
segundo.idade = 10;
primeiro = &segundo;
```

- Para acessar o campo idade de primeiro:

```
(*primeiro).idade
primeiro -> idade
```

Vetores, matrizes e estruturas dentro de estruturas

- Os elementos das estruturas podem ser simples ou complexos, assim, podemos colocar vetores, matrizes e até estruturas dentro das estruturas. Veja:

```
struct complexa {
   char setor[21];
   struct pessoa funcionarios[50];
}
```

Campos de bit

- *C possui metodos para acessar somente um bit dentro de um byte. Isto é útil para:*
  - *Economizar memória declarando várias variáveis booleanas num só byte*
  - *Comunicar com dispositivos que transmitem informações diversas codificadas em bytes*
  - *Rotinas de codificação que precisam acessar bits dos bytes*

Como declarar campos de bit

- Os campos de bit só podem ser declarados dentro de estruturas.
- Forma geral:

```
struct nome_estrutura {
    tipo1 var1 : comprimento;
    tipo2 var2 : comprimento;
} nome_var ;
```

- Os tipos podem ser: int, signed e unsigned. Quando o tamanho é 1, ele deve ser obrigatoriamente unsigned.

Exemplo

```
struct dispositivo {
   unsigned ativo : 1;
   unsigned pronto : 1;
   unsignet erro : 1;
   unsigned : 2;
   unsigned ultimo_erro : 3;
}
```

Uniões

- *Em C, uma união é uma localização de memória que é usada por muitas variáveis diferentes, que podem ser de tipos diferentes.*
- *A declaração e o acesso a uma união é feita de forma semelhante as estruturas, só que usando a palavra reservada union no lugar de struct.*

Exemplo

```
union dois_bytes {
   int inteiro;
   char caracter[2];
} valor;
main()
{
   valor.inteiro = 'A' * 256 + 'B';
   printf("%c %c", valor.caracter[0],
   valor.caracter[1]);
}
```

Enumerações

- Enumeração é um conjunto de constantes inteiras com nome e e especifica todos os valores legais que uma variável daquele tipo pode ter.
- Para declarar:

  enum nome_tipo { lista de constantes } nome_var;

- Todas as constantes receberão valores inteiros a começar por zero, a não ser que seja especificado o contrário.

Exemplo

```
enum tamanhos {pequeno, medio, grande =
   5} tamanho;
main()
{
   tamanho = pequeno; printf("%d", tamanho);
   tamanho = medio; printf("%d", tamanho);
   tamanho = grande; printf("%d", tamanho);
}
```

sizeof

- *Como estamos trabalhando só com computadores IBM PC, até agora, não nos preocupamos com os tamanhos das variáveis pois já sabemos seus valores.*
- *C possui o operador sizeof com a finalidade de garantir a portabilidade dos programas entre ambientes cujos tamanhos das variáveis sejam diferentes.*
- *sizeof retorna a quantidade de bytes que uma variável ocupa na memória.*

Exemplos

- *Alocar memória para 10 floats:*

  *float *numeros;*

  *numeros = (float *) calloc(10, sizeof(float));*

- *Imprimir o tamanho gasto por uma variável inteira:*

  printf("%d", sizeof(int));

typedef

- *C permite que sejam definidos explicitamente novos tipos de dados usando a palavra reservada typedef.*
- *typedef não cria realmente uma nova classe de dados, mas sim define um novo nome para uma tipo já existente.*
- *O uso do typedef torna os programas em C mais legíveis e mais portáteis pois bastará alterar a definição do typedef quando trocar de ambiente.*

Exemplo

- *Após as definições:*

  typedef float balanco;

  typedef char string[21];

  typedef struct pessoa tipo_pessoa;

- *As seguintes declarações serão equivalentes:*

Operadores avançados

- *C possui vários operadores especiais que aumentam em muito sua força e flexibilidade - especialmente na programação a nível de sistema.*

Operadores bit a bit

- Como C foi projetada para substituir a linguagem assembly na maioria das tarefas de programação, ela possui um completo arsenal de operadores bit a bit.
- Os operadores bit a bit só podem ser usados nos tipos char e int.

Tabela dos operadores bit a bit

Operador ?

- O operador ? pode ser usado para substituir comandos if / else que tenham a forma geral:

```
if (condição)
   expressão1;
else
   expressão2;
```

- Forma geral:

```
condição ? expressão1 : expressão 2;
```

Exemplo

- *A seqüência de comandos:*

```
x = 10;
if (x > 9) y = 100;
else y = 200;
```

- *pode ser substituída por:*

```
x = 10;
y = x > 9 ? 100 : 200;
```

Formas abreviadas

- C permite que sejam escritas formas abreviadas dos comandos de atribuição. Os comandos:

```
x = x + 10;
y = y * 20;
z = z - 5;
```

- podem ser reescritos da forma:

```
x += 10;
y *= 20;
z -= 5;
```

Operador vírgula

- O operador vírgula é usado para juntar várias expressões. O compilador sempre avalia o lado esquerdo da vírgula como void. Assim, a expressão do lado direito ficará sendo o valor de toda expressão separada por vírgula.
- Por exemplo:

  x = (y = 3, y + 1);
- Atribuirá 3 a y e 4 a x.

Parênteses e colchetes

- *C considera os parênteses e os colchetes como operadores.*
- *Os parênteses executam a tarefa esperada de aumentar a precedência dos operadores que estão dentro deles.*
- *Os colchetes executam a indexação de vetores e matrizes.*

Resumo de precedência

Funções comuns

- *A biblioteca padrão de funções da linguagem C é muito ampla.*
- *Os programas em C fazem uso intenso das funções da biblioteca.*
- *Os programadores iniciantes tendem a reescrever funções já existentes nas bibliotecas.*

Funções matemáticas

- Os protótipos das funções matemáticas ficam no arquivo math.h. Veja alguns protótipos:

```
double sin(double arg);
double cos(double arg);
double tan(double arg)
double exp(double arg);
double log(double num);
double log10(double num);
double pow(double base, double exp);
double sqrt(double num);
```

Alguns erros comuns de programação

- *Por ser uma linguagem que dá muito poder ao programador, também é muito fácil de errar em C.*
- *Como o compilador aceita praticamente tudo o que se escreve, o programador deve ter atenção redobrada na hora de programar.*

Erros de ordem de processamento

- *Os operadores de incremento e decremento são usados na maioria dos programas em C, e a ordem de ocorrência da operação é afetada pelo fato de esses operadores precederem ou sucederem a variável. Logo, se y=10*

  x = y++;

- *será diferente de*

  x = ++y;

Problemas com ponteiros

- O mau-uso de ponteiros em C pode causar muitos problemas. Veja os dois exemplos ERRADOS abaixo:

```
int *x;
char *p;
*p = malloc(100);
*x = 10;
```

Redefinindo funções

- *É possível, mas não recomendado, que seja criada uma função com o mesmo nome de uma existente na biblioteca de C. Isto fará com que qualquer chamada a esta função seja direcionada para a nova.*
- *A pior parte é que, mesmo o nosso programa não referenciando uma função da biblioteca, as próprias funções da biblioteca podem estar se referenciando, o que causará problemas da mesma forma. Por exemplo, o programador reescreveu a função getc mas esqueceu-se que a função scanf a chama.*

Erros por um

- Todos os índices dos vetores e matrizes em C começam em 0. Logo, o programa abaixo está errado.

```
main()
{
   int x, num[100];
   for (x = 1; x <= 100; x ++)
     num[x] = x;
}
```

Erros de limite

- Muitas funções em C (inclusive as das bibliotecas) não possuem (ou possuem pouca) verificação de limites. Assim, a chamada a gets abaixo pode gerar um problema caso o usuário digite mais que 20 caracteres.

```
main()
{  char texto[21];
   gets(texto);
}
```

Omissões de declaração de função

- *Esquecer de definir o protótipo de uma função pode causar erro pois o compilador estará esperando sempre que as funções não declaradas retornem um inteiro. O mesmo problema ocorre também com os parâmetros.*

Erros de argumentos de chamada

- Os argumentos passados a uma função devem ser do mesmo tipo do esperado por elas. O programa abaixo está errado:

```
main()
{
   int x;
   char string[10];
   scanf("%d%s", x, string);
}
```

Exercício (parte 1)

- Faça um programa que leia 5 nomes com suas respectivas idades e mostre o maior e o menor.
- Vamos dividir o problema em partes, primeiro vamos definir uma estrutura contendo um campo nome e um campo idade:

```
typedef struct {
    char nome[21];
    int idade; } pessoa;
```

Exercício (parte 2)

- Vamos definir a função mais_velho que recebe 2 ponteiros do tipo pessoa e retorna um ponteiro para a pessoa mais velha:

```
pessoa *mais_velho(pessoa *p1, pessoa
   *p2);
{
   if (p1 -> idade < p2 -> idade) return p2;
   else return p1;
}
```

Exercício (parte 3)

- Vamos definir a função mais_novo que recebe 2 ponteiros do tipo pessoa e retorna um ponteiro para a pessoa mais nova:

```
pessoa *mais_novo(pessoa *p1, pessoa
   *p2);
{
   if (p1 -> idade > p2 -> idade) return p2;
   else return p1;
}
```

Exercício (parte 4)

- Vamos definir a função le_pessoa que recebe um ponteiro do tipo pessoa e faz a leitura do teclado:

```
void le_pessoa(pessoa *p)
{
   printf("Nome: ");
   gets(p -> nome);
   printf("Idade: ");
   scanf("%d", p -> idade);
}
```

Exercício (parte 5)

- Vamos definir a função mostra_pessoa que recebe um ponteiro para uma pessoa e mostra seus dados na tela.

```
void mostra_pessoa(pessoa *p)
{
   printf("Nome: %s\nIdade: %d\n", p->nome,
   p->idade);
}
```

Exercício (parte 6)

- *Agora só falta declarar as variáveis e definir o programa principal, que será composto por um loop de entrada, um loop de pesquisa e a apresentação dos resultados.*
- *Dividindo o código do programa, fica mais fácil resolver os problemas e verificar os erros que possam ocorrer.*

Exercício (parte 7)

```
main()
{  pessoa pessoas[5], *velho, *novo;
   int i;
   for(i = 0; i < 5; i++) le_pessoa(&pessoas[i]);
   velho = novo = pessoas;
   for(i = 1; i < 5; i++) {
     velho = mais_velho(velho, &pessoas[i]);
     novo = mais_novo(novo, &pessoas[i]);
   }
   puts("O mais novo é\n"); mostra_pessoa(novo);
   puts("O mais velho é\n"); mostra_pessoa(velho);
}
```