P R O G R A M A Ç Ã O

E M

L I N G U A G E M C / C++

Por Marcos Romero

http://romerogames.blogspot.com

SUMÁRIO

Parte I - Conhecendo a linguagem C

Capítulo 1 : Primeiros programas em C

- Programa 1:

Começaremos nosso estudo da linguagem C examinando um programa bem simples que escreve na tela o texto "Alo Turma":

```
/* Nome do arquivo : cap1_1.c */
main()
{
  printf("Alo Turma");
}
```

A primeira linha do programa é apenas um comentário indicando o nome do arquivo. Na linguagem C os comentários são delimitados por /* e */. O arquivo que contém o código deve ter a extensão ".C". Em seguida está a função "main( )" que todo programa em C deve ter. É nesta função que o programa inicia sua execução. O par de chaves { } delimitam um bloco de código que, neste caso, é o conteúdo da função "main( )".

A função "printf( )" é usada para escrever texto na tela. Ela recebe como argumento uma string (conjunto de caracteres) que está entre aspas duplas "...". Depois do printf há um ponto-e-vírgula que indica o fim de um comando em C.

É importante observar que a linguagem C diferencia as letras maiúsculas das minúsculas, isto significa que se for escrito "Main( )" ou "MAIN( )" um erro ocorrerá.

- Programa 2:

```
/* Nome do arquivo : cap1_2.c */
main()
{
  int nota;
  nota = 8;
  printf("\nA nota da prova foi %d \n", nota);
  printf("FIM DE PROGRAMA");
}
```

Este programa introduz um conceito muito importante: as variáveis. Elas são usadas para armazenar valores. No exemplo, o comando "int nota;" define uma variável que guardará valores do tipo inteiro. A linha seguinte atribui o valor 8 à variável. No C o símbolo "=" é o operador de atribuição e sua função é de pegar o valor que está a sua direita e armazenar na variável que está a sua esquerda.

A função printf( ) usa o código de formatação %d dentro da string para indicar que um inteiro em formato decimal será escrito nesta posição. A variável é passada como segundo argumento, é usada uma vírgula para separar os argumentos. O "\n" também é um código de formatação conhecido como *quebra-de-linha* e faz o cursor de inserção do texto ir para a linha seguinte.

- Programa 3:

```
/* Nome do arquivo : cap1_3.c */
main()
{
  int num1,num2,resultado;
  printf("\nDigite um numero de 1 a 9 e pressione ENTER : ");
  scanf("%d",&num1);
  printf("\nDigite outro numero de 1 a 9 e pressione ENTER : ");
  scanf("%d",&num2);
  resultado = num1 + num2;
  printf("\n %d + %d = %d", num1, num2, resultado);
}
```

Pode-se definir várias variáveis do mesmo tipo em uma única linha como ivsto neste exemplo. É usado a função scanf( ) para ler um valor digitado pelo usuário. O código %d indica que será lido um inteiro e armazenado na variável "num1". O "&" que precede o "num1" é sempre necessário para que um valor possa ser guardado em uma variável, mais adiante no curso entenderemos melhor o motivo.

Depois da leitura do segundo número há uma expressão que soma os dois números lidos e armazena o valor resultante na variável "resultado". Finalmente, é usado um printf( ) para ilustrar a expressão. Cada "%d" corresponde a um argumento na ordem em que aparecem.

Dica : O esquecimento do "&" na função scanf( ) é uma fonte muito comum de erros, principalmente porque o compilador não irá lhe avisar quando isto ocorrer.

Capítulo 2 : Variáveis e Operadores

- Variáveis:

Há algumas regras básicas para a criação de nomes de variáveis :

- O primeiro caractere deve ser uma letra ou um sublinhado, mas não um número.
- Os caracteres seguintes podem ser números, letras ou sublinhados.
- Não podem ser usados nenhum dos operadores da linguagem C.
- Não podem ser usados caracteres acentuados.

Exemplos:

corretos	**incorretos**
aluno1	1aluno
calculo	cálculo
endereco	endereço
nome_aluno	nome-aluno

Dica: Lembre-se que a linguagem C distingue os caracteres maiúsculos dos minúsculos, então se for definida uma variável de nome "NOTA" e depois houver uma referência como "nota", o compilador não associará as duas e ocorrerá um erro.

A tabela seguinte contém os tipos de dados básicos do C.

Tipo	**Bits**	**Faixa**
char	8	-128 a 127
int	16	-32768 a 32767
float	32	3.4 E-38 a 3.4 E+38
double	64	1.7 E-308 a 1.7 E+308

O tipo **char** é usado para guardar caracteres do padrão ASCII de 8 bits que contém letras, dígitos e símbolos. As variáveis do tipo **int** podem conter valores inteiros. O tipo **float** é necessário para armazenar números fracionários que também são chamados de ponto flutuante. O tipo **double** eqüivale ao float só que tem uma capacidade bem maior de armazenamento.

Pode-se aplicar o modificador de tipo **unsigned** antes de **char** e **int**, fazendo com que as variáveis sé aceitem valores positivos. O tipo **char** passa a aceitar valores entre 0 e 255 e o tipo **int** entre 0 e 65535. O tipo **char** pode ser usado também como se fosse um inteiro pequeno. Os modificadores **long** e **short** são usados com o tipo **int**. O **long** faz com que a variável tenha 32 bits para armazenamento enquanto o short tem apenas 16 bits. Nos compiladores para sistemas operacionais de 32 bits (como o Windows 95) o tamanho padrão do tipo **int** é de 32 bits.

O programa a seguir ilustra o uso das variáveis:

```
/* cap2_1.c */

main()
{
  char letra = 'm';
  int num_inteiro;
  float num_real;

  unsigned int sem_sinal;
  short int num_16b;
  long int num_32b;

  printf("\n *** ESTUDO DAS VARIAVEIS *** \n");

  printf("\n Digite um valor inteiro e um valor fracionario :\n");
  scanf("%d %f", &num_inteiro, &num_real);

  printf("\n O no. inteiro digitado foi : %d", num_inteiro);
  printf("\n O no. fracionario digitado foi : %f", num_real);
  printf("\n No. fracionario com formatacao : %.3f", num_real);

  printf("\n\n A letra \'%c\' tem o codigo ASCII de %d",letra,letra);

  sem_sinal = -10; /* esta variavel nao suporta no. negativos*/
  printf("\n\n Valor da variavel sem_sinal : %u", sem_sinal);

  num_16b = 35000; /*este valor excede o limite maximo da variavel*/
  num_32b = 35000;/* sem problemas*/

  printf("\n Valor do num_16b : %d", num_16b);
  printf("\n Valor do num_32b (long): %ld", num_32b);

}
```

Logo no começo do programa é definida uma variável do tipo **char** que recebe uma letra inicial. Um único caractere deve ter aspas simples (ex: 'a') enquanto que uma string tem aspas duplas (ex: "a"). Em seguida são definidas outras variáveis de diversos tipos.

É usado outros códigos de formatação do **printf( )** no programa. Na lista a seguir há comentários sobre os códigos mais comuns.

código	**significado**
%c	caractere
%d	inteiro decimal
%f	número fracionário
%u	inteiro sem sinal
%ld	inteiro longo
%%	imprime o sinal %
\n	mudança de linha
\t	tabulação
\"	imprime aspas duplas
\'	imprime aspas simples
\\	imprime a barra invertida

No programa a função **scanf( )** está lendo dois valores de uma única vez. Para separar os valores na digitação pode-se usar um espaço, um tab ou uma quebra-de-linha (tecla ENTER). Mais adiante é visto em um printf o código "%.3f" que especifica um número fracionário de 3 casas decimais.

A linha seguinte demonstra que a variável do tipo **char** pode ser usada como caractere ou como um valor decimal. Depois disso é feito um teste com os limites das variáveis.

- Operadores:

Um operador é um símbolo que instrui o compilador a executar certas manipulações matemáticas ou lógicas. A lista a seguir mostra os operadores aritméticos.

Operador	**Ação**
+	adição
-	subtração
*	multiplicação
/	divisão
%	resto da divisão

Estes operadores executam as operações básicas da matemática. Deve-se ter cuidado com o operador de divisão, se ele for usado com números inteiros, seu resultado também será inteiro (Ex: 9 / 4 será igual a 2). O operador % retorna o resto de uma divisão entre inteiros (Ex: 9 % 4 será igual a 1).

Há também os operadores de relação e os operadores lógicos, que retornam sempre falso (0) ou verdadeiro (1). Em C, o verdadeiro é representado por qualquer valor diferente de zero, enquanto que o falso é zero.

Operadores de Relação

Operador	**Ação**
>	maior que
>=	maior ou igual a
<	menor que
<=	menor ou igual a
= =	igual a
!=	diferente de

Operadores Lógicos

Operador	**Ação**
&&	AND (E)
| |	OR (OU)
!	NOT (NÃO)

Dica: É comum confundir o operador de relação "= =", que compara dois valores e retorna verdadeiro (1) se forem iguais ou falso (0) se forem diferentes, com o operador de atribuição "=" que pega o valor que está à sua direita e armazena na variável à sua esquerda.

Os operadores bit a bit são operadores avançados geralmente utilizados em programação de baixo nível. Eles trabalham diretamente com o número em sua forma binária, só podem ser usados com os tipos char e int.

Operador	**Ação**
&	AND (E)
|	OR (OU)
^	XOR (OU exclusivo)
~	NOT (NÃO)
>>	Deslocamento para a direita
<<	Deslocamento para a esquerda

A linguagem C possui algumas formas abreviadas de operadores como o de incremento (++) que soma um ao seu operando e operador de decremento (--) que subtrai um. Ex: num++; num--;

Outra forma de abreviação é demonstrada a seguir:

```
x = x + 10;
```

Pode ser reescrito como :

```
x += 10;
```

Este formato pode ser usado com os operadores aritméticos do C.

Ex: x-=5; x /= 3;

Pode-se forçar uma expressão a ser de um tipo usando uma construção chamada *cast*. Por exemplo, se x for um inteiro e você quer garantir que o resultado da expressão x / 3 seja um float (fracionário), use o seguinte comando :

```
(float) x / 3;
```

Há outro operador avançado conhecido como operador ternário (?:).

Ex: y = x < 10 ? 100 : 200 ;

Se a expressão (x < 10) for verdadeira então o valor resultante será o 100, se a expressão for falsa, o valor resultante será o 200.

A lista a seguir mostra a ordem de precedência que os operadores têm em uma expressão.

Precedência dos operadores

Maior

```
! ~ ++ --  - (cast)
*  /  %
+  -
<<  >>
<  <=  >  >=
= =  !=
&
^
|
&&
| |
?:
=  +=  -=  *=  /=
```

Menor

Use sempre parênteses para deixar clara a ordem de execução de uma expressão. Por exemplo, compare as seguintes expressões :

```
x = m*2 + 3*j / f - 5;

x = ( m*2 ) + ( (3 * j / f ) - 5 );
```

```
/*
      cap2_2.c
      Autor: Marcos Romero
*/

main()
{
  int x,y;
  int resto1, resto2;

  printf("\n *** ESTUDO DOS OPERADORES *** \n");

  printf("\n Digite dois numeros inteiros:\n");
  scanf("%d %d", &x, &y);

  printf("\n\n Operadores Aritmeticos ");
  printf("\n %d + %d = %d ", x, y, x+y);
  printf("\n %d - %d = %d ", x, y, x-y);
  printf("\n %d * %d = %d ", x, y, x*y);
  printf("\n %d / %d = %d (divisao inteira)", x, y, x/y);
  printf("\n %d / %d = %.2f (divisao fracionaria)", x, y,(float) x/y);
  printf("\n %d %% %d = %d (resto da divisao)", x, y, x%y);

  printf("\n\n Operadores de Relacao ");
  printf("\n %d == %d retorna %d", x, y, x==y);
  printf("\n %d != %d retorna %d", x, y, x!=y);
  printf("\n %d <  %d retorna %d", x, y, x<y);
  printf("\n %d >  %d retorna %d", x, y, x>y);

  resto1 = x % 2; /* o resultado sera 0 ou 1 */
  resto2 = y % 2;

  printf("\n\n Operadores Logicos ");
  printf("\n%d && %d retorna %d -(AND)",resto1,resto2,resto1&&resto2);
  printf("\n%d || %d retorna %d -(OR)",resto1,resto2,resto1 ||resto2);
  printf("\n ! %d retorna %d -(NOT)", resto1, !resto1);

  printf("\n\n Uso de Parenteses ");
  printf("\n %d + %d * 4 = %d", x, y, x + y * 4);
  printf("\n(%d + %d)* 4 = %d", x, y, (x + y) * 4);

  printf("\n\n Resultado da expressao: %d<=5 ? 500:1000 e igual a %d",
            x, x <= 5 ? 500:1000);
}
```

Capítulo 3: Controle de fluxo

- O comando if:

O comando **if** (se) é usado para que um código só seja executado mediante uma condição ser verdadeira (qualquer valor diferente de 0).

Ex: if (x < 10) printf("%d é menor que 10", x);

Há também a cláusula **else** (senão) que é usada em conjunto com o if para que um código seja executada caso a condição seja falsa (0).

Ex: if(senha = = 123) printf("Acesso concedido");
else printf("Acesso negado");

Se for necessário associar mais de um comando a uma condição if (ou else) pode-se usar um bloco de código, que é delimitado por chaves { }. A expressão condicional do if pode ser de diversas formas, bastando que o resultado seja um valor zero ou não-zero, por exemplo, o código a seguir só imprimirá a divisão se o denominador for diferente de zero.

Ex: if(y) printf(" %d", x / y);

```
/* cap3_1.c */
main()
{
  int nota;

  printf("\nDigite a nota da prova (0-10): ");
  scanf("%d",&nota);

  if(nota == 10)
  {
      printf("\n PARABENS. Aprovado com nota maxima.");
  }
  else if(nota >= 7 && nota < 10)
  {
      printf("\n Aprovado com conceito BOM");
  }
  else if(nota >= 5 && nota < 7)
  {
      printf("\n Aprovado com conceito REGULAR");
  }
  else if(nota >= 0 && nota < 5)
  {
      printf("\n Reprovado");
  }
  else
  {
      printf("Nota invalida");
  }
}
```

O comando switch :

O comando **switch** (chave) testa o valor de uma variável contra uma lista de constantes inteiras ou de caracteres. Se um valor igual for encontrado então o bloco de código associado a este valor será executado.

A forma do comando switch é :

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

O comando **default** é opcional e é executado quando não for encontrado nenhum valor igual. O comando **break** é usado para indicar o fim da seqüência de comandos de cada **case**.

```
/* cap3_2.c */
main()
{
  int escolha;
  int x,y;
  printf("\n Digite dois numeros inteiros:\n");
  scanf("%d %d", &x, &y);
  printf("\n 1. Adicao");
  printf("\n 2. Subtracao");
  printf("\n 3. Multiplicacao");
  printf("\n 4. Divisao");
  printf("\n Escolha a operacao: ");
  scanf("%d",&escolha);

  switch(escolha)
  {
      case 1:
                  printf("\n Adicao = %d",x + y);
                  break;
      case 2:
                  printf("\n Subtracao = %d",x - y);
                  break;
      case 3:
                  printf("\n Multiplicacao = %d",x * y);
                  break;
      case 4:
                  /*testar se o denominador nao e zero*/
                  if(y) printf("\n Divisao : %.2f", (float) x / y);
                  else printf("\n Divisao por zero");
                  break;
      default:
                  printf("\n Operacao invalida");
  }
}
```

- O laço for :

Os laços na linguagem C são usados para repetir um conjunto de instruções até que uma certa condição seja satisfeita. O comando for é um dos tipos de laço, um exemplo de uso está a seguir:

```
for(x = 1; x <=20; x++) printf("\n %d", x);
```

Este exemplo escreverá na tela os números de 1 a 20.

O comando **for** tem três partes principais: inicialização, condição, incremento. No exemplo, a inicialização é o "x = 1", que atribui um valor inicial à variável de controle do laço (x). A condição é o "x <= 20", enquanto esta condição for verdadeira o laço continuará executando, o seu teste é feito no início do laço. A última parte do for é o incremento que no exemplo é "x++", esta parte define como será alterada a variável de controle do laço.

```
/* cap3_3.c */

main()
{
  int min, max, soma = 0, i;

  printf("\n*** Este programa soma uma faixa de valores ***");
  printf("\n\nInforme o valor minimo e maximo:\n");
  scanf("%d %d", &min, &max);

  if(max < min)
  {
      printf("O valor maximo deve ser maior ou igual que o minimo");
  }
  else
  {
      for(i = min; i <= max; i++)
      {
        printf("\n Valor: %d", i);
        soma += i;
        printf("\t Soma: %d", soma);
      }
      printf("\n\n Resultado final: %d", soma);
  }
}
```

- Os laços while e do...while :

Há outro tipo de laço disponível no C chamado **while** (enquanto). Ele tem o seguinte formato:

while(*condição*) *comando*

Enquanto a *condição* for verdadeira o *comando* será executado, onde *comando* pode ser um único comando ou um bloco de código.

Outra forma de se definir um laço deste tipo é usando o do...while :

do {
comandos
} while (*condição*) ;

A única diferença desta forma para a anterior é que a condição é testada no final, fazendo com que os comandos sejam executados ao menos uma vez.

```
/* cap3_4.c */

main()
{
  int x;

  printf("\n *** CALCULO DO CUBO DE UM NUMERO ***\n");

  do {
       printf("\n Digite um numero (ou 0 p/ sair) : ");
       scanf("%d", &x);

       printf("\nSeu cubo e : %d \n", x*x*x);

  } while(x != 0);
}
```

- Interrupção de laços:

Pode-se usar o comando **break** para interromper imediatamente um laço. Ele é geralmente usado quando algum evento extra tem influência no encerramento do laço.

A linguagem C permite o uso de laços infinitos. Ex: for(; ;) ou while(1) . A única forma de terminá-los é usando o comando **break**.

Outro comando utilizado dentro de laços é o **continue**. Ele faz com que o laço vá para a próxima iteração sem executar o restante do código.

```
/* cap3_5.c */

main()
{
  char letra;
  int i;

  printf("\n Exemplo de uso do continue \n");

  printf("\n Imprime os pares \n");

  for(i=0; i<10; i++)
  {
      if(i % 2) continue;  /*testa se o numero e impar */

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

  printf("\n\n Exemplo de uso do break \n");

  printf("\n Valores ASCII das letras \n");

  while(1) /*laco infinito  */
  {
      printf("\nDigite uma letra (ou '.' p/ sair) : ");

      scanf("\n%c",&letra);

      if(letra == '.') break;

      printf("\nA letra %c tem codigo ASCII: %d \n", letra, letra);
  }
}
```

Capítulo 4: Matrizes e Strings

- Matrizes:

Uma matriz é usada para guardar um conjunto de variáveis de mesmo tipo usando um único nome. Um índice é usado para acessar um elemento da matriz.

Ex: int notas[10];

O número entre colchetes indica a quantidade de elementos que a matriz tem. Em C, todas as matrizes **começam em zero**, então no exemplo o primeiro elemento é notas[0] e o último é notas[9]. Para acessar um dos elementos da matriz basta colocar o seu índice entre os colchetes. O exemplo a seguir preenche a matriz com os valores de 0 a 9.

Ex: for(i = 0; i < 10; i++) notas[i] = i ;

A linguagem C não faz verificação de limites em matrizes, por isso é possível atribuir valores à posições que não pertencem a matriz. Este ato alterará de forma indevida a memória do sistema e pode gerar erros na execução do programa.

Ex: notas[12] = 5 ; /* o índice máximo é 9 */

```
/* cap4_1.c */

main()
{
  int i;
  int notas[4];

  printf("\n Digite as notas: \n");

  for(i = 0; i < 4; i++)
  {
      printf("Nota %d: ", i + 1);
      scanf("%d", &notas[i]);
  }

  printf("\n\n As notas em ordem inversa: \n");

  for(i = 3; i >= 0; i--)
  {
      printf("\n Nota %d = %d ", i + 1, notas[i]);
  }
}
```

- Strings :

A linguagem C não possui um tipo interno definido como string, para usá-la em C é necessário criar uma matriz de caracteres. Uma string é finalizada por um zero representado como '\0', por isso é preciso definir o tamanho de uma string como tendo um espaço a mais para armazenar o zero. Por exemplo, para declarar uma string que terá 8 caracteres, use:

```
char  texto[9];
```

Para ler uma string pelo teclado é utilizada uma função da biblioteca de C chamada **gets( )**. O parâmetro é o nome da matriz sem qualquer índice.

Ex: gets(texto);

Para escrevê-la na tela pode-se usar a função printf( ) com o código de formatação **%s** e o nome da matriz sem qualquer índice.

Ex: printf("%s", texto);

A linguagem C define diversas funções para manipulação de strings. A função **strcpy( )** é usada para copiar o conteúdo de uma string para outra, enquanto que a função **strcat( )** anexa uma string ao final da outra. A função **strcmp( )** compara duas strings e retorna zero se forem iguais, se a primeira string for maior então o resultado será um valor positivo, se a segunda string for maior o resultado será negativo. Outra função de uso comum é a **strlen( )** que retorna o comprimento da string.

```
/* cap4_2.c */

main()
{
  char nome[21], frase[40] ;

  do {
      printf("\n Para finalizar o programa digite \"sair\" \n");

      printf("\n Digite seu nome (max=20 letras): ");
      gets(nome);

      strcpy(frase, "\n bom dia ");
      strcat(frase, nome);

      printf("%s", frase);
      printf("\n O nome %s tem %d letras \n",nome, strlen(nome) );

  } while (strcmp(nome,"sair") != 0);
}
```

- Matrizes de duas dimensões

Uma matriz de duas dimensões tem uma forma parecida a uma tabela. Para defini-la em C use dois índices.

Ex: int mat2d[3][4] ;

Para acessar cada posição da matriz basta especificar os dois índices devidamente.

Ex: mat2d[1][1] = 5 ;

Um uso comum para a matriz de duas dimensões é como matrizes de strings.

Ex: char mat_string[10][50];

O segundo índice (50) está relacionado ao tamanho de uma string enquanto que o primeiro índice indica a quantiade de strings. Para ler uma string e guardá-la em uma posição da matriz, deve-se passar como parâmetro para a função **gets( )** o nome da matriz somente com o índice esquerdo.

Ex: gets(mat_string[3]);

É permitido inicializar as matrizes, desde que elas sejam globais (este conceito será visto mais adiante no curso), ou seja ela é definida antes da função **main( )**. Pode-se deixar de especificar o índice fazendo com que o C automaticamente dimensione a matriz.

Ex: int pares[5] = { 0, 2, 4, 6, 8 };
char msg[ ] = "Seja bem vindo" ;

```
/* cap4_3.c */

/*string global */
char msg[] = "Digite 3 nomes: ";

main()
{
  char nomes[3][50];
  int i;

  printf("\n %s \n",msg);

  for(i = 0; i < 3; i++) gets(nomes[i]);

  printf("\n Nomes digitados: \n");

  for(i = 0; i < 3; i++)
  {
      printf("\n Nome %d = %s ", i+1, nomes[i]);
  }
  printf("\n\n A 3a. letra do 1o. nome = %c",nomes[0][2]);
  printf("\n A 4a. letra do 2o. nome = %c",nomes[1][3]);
  printf("\n A 2a. letra do 3o. nome = %c",nomes[2][1]);

}
```

- Programa de exemplo da Parte I

Este programa usa uma forma bem simples de criptografia que apenas alterar o valor ASCII de cada letra. Há duas matrizes de strings que guardam as palavras originais e as criptografadas. Elas ficam armazenadas na memória para que sejam listadas posteriormente. No começo do programa é usado o comando **#define** para definir constantes.

```
/*
Nome    : cripto.c
Autor   : Marcos Romero
Programa de exemplo da parte I
*/

/* valores constantes */
#define TAM_PALAVRA  50
#define MAX_PALAVRAS 30
#define CHAVE     5

main()
{
  int n_palavras = 0, escolha, parar;
  int i , j;
  char palavras[MAX_PALAVRAS][TAM_PALAVRA];
  char cripto[MAX_PALAVRAS][TAM_PALAVRA];

  printf("\n\n *** EXEMPLO DE CRIPTOGRAFIA *** \n");

  while(1)
  {
      printf("\n\n M E N U \n");
      printf("\n 1. Inserir palavras");
      printf("\n 2. Listar palavras");
      printf("\n 3. Encerrar programa");

      printf("\n\n Escolha: ");
      scanf("%d%*c", &escolha);

      switch(escolha)
      {
        case 1:
            /* testar limites de palavras */
            if(n_palavras >= MAX_PALAVRAS)
            {
              printf("\n Alcancou limite de palavras.");
              exit(0); /*encerra o programa */
            }

            printf("\n\n Digite \"sair\" para voltar ao menu\n\n");
            parar = 0;
            do {
             printf("\n Digite a palavra a ser criptografada:\n");
             gets(palavras[n_palavras]);
```

```
              /*passar palavra p/ minuscula */
          for(i=0; palavras[n_palavras][i] ; i++)
          {
           palavras[n_palavras][i]= tolower(palavras[n_palavras] [i]);
          }

          if( strcmp(palavras[n_palavras],"sair") == 0 )
          {
              parar = 1;
          }
          else
          {
              /*copiar a palavra*/
              strcpy( cripto[n_palavras], palavras[n_palavras] );

              /*criptografar*/
              for(i=0; cripto[n_palavras][i] ; i++)
              {
                cripto[n_palavras][i] = cripto[n_palavras][i] + CHAVE;
              }

              printf("\nCriptografia : %s \n",cripto[n_palavras]);

              n_palavras++; /*atualiza o contador de palavras*/

          }/*fim else*/

      } while(!parar);
      break; /*fim case 1*/

  case 2:
            if(n_palavras == 0) printf("\n\n Nao ha palavras \n");
            else
            {
              printf("\n Palavras Criptografadas \n\n");
              printf("\n ORIGINAL \t CRIPTOGRAFADA\n");
              for(i=0; i< n_palavras; i++)
              {
                  printf("\n %8s \t %s",palavras[i], cripto[i]);
              }
              printf("\n\n FIM DA LISTA \n\n");
            }
            break; /*fim case 2*/

  case 3:
            exit(0);
            break;

  default: printf("\n Opcao invalida\n ");
             break;

      }/*fim switch*/

  } /*fim while*/

}/*fim main*/
```

PARTE I I - Se aprofundando na linguagem C.

Capítulo 5 : Estruturas

A estrutura é usada para manter juntas diversas variáveis que tem informações relacionadas. A definição de uma estrutura forma um gabarito que é utilizado para criar variáveis de estrutura.

Ex:

```
struct info {
      char titulo[30];
      char autor[30];
      int  edicao;
};
```

Dica: Depois da chave que fecha a definição da estrutura é preciso colocar o ponto e vírgula.

Para criar uma variável com essa estrutura use:

```
struct info livro;
```

Para acessar os elementos individuais da estrutura, utiliza-se o '.' após o nome da variável de estrutura e em seguida o nome da variável interna que deseja acessar.

Ex:
```
livro.edicao = 1;
strcpy( livro.titulo , "Programando em C");
```

```
/* cap5_1.c */

struct retangulo {
            int largura;
            int altura;
            int area;
            };
main()
{
      struct retangulo ret1, ret2;

      printf("\nDigite a largura do 1o. retangulo : ");
      scanf("%d",&ret1.largura);
      printf("\nDigite a altura do 1o. retangulo : ");
      scanf("%d",&ret1.altura);
      ret1.area = ret1.largura * ret1.altura;

      printf("\nDigite a largura do 2o. retangulo : ");
      scanf("%d",&ret2.largura);
      printf("\nDigite a altura do 2o. retangulo : ");
      scanf("%d",&ret2.altura);
      ret2.area = ret2.largura * ret2.altura;
```

```
      printf("\n\nArea do 1o. retangulo = %d", ret1.area);
      printf("\n\nArea do 2o. retangulo = %d \n", ret2.area);

      if(ret1.area == ret2.area)
      {
            printf("\nAs areas dos retangulos sao iguais.\n");
      }
      else if(ret1.area > ret2.area)
      {
            printf("\nA area do 1o. retangulo e maior.\n");
      }
      else
      {
            printf("\nA area do 2o. retangulo e maior.\n");
      }

}
```

É muito comum o uso de matrizes de estrutura. Por exemplo, para declarar uma matriz de 20 livros que tem o formato da estrutura "info" definido anteriormente, use:

struct info livros[20];

Para acessar um elemento de um dos itens da matriz basta indexar o nome da estrutura e depois colocar o '.' com o nome do elemento.

Ex: printf("%s" , livros[2].titulo) ;

Dentro de uma estrutura pode haver matrizes e outras estruturas. Veja o seguinte exemplo :

```
struct  _end {
      char  rua[40];
      int numero;
};
struct _trab {
      char nome[40];
      int salario;
      struct  _end  endereco;
};

struct  _trab   trabalhadores;

trabalhadores.endereco.numero = 1000;
```

Há um comando na linguagem C chamado **typedef** que é usado para definir um novo nome para um tipo existente. Se você quiser criar um novo nome para float, e depois definir uma variável com este novo nome, use:

typedef float indice ;

indice juros;

O typedef é muito utilizado para criar nomes para estruturas, facilitando as definições de novas variáveis. Por convenção, o novo tipo criado usa letras maiúsculas para diferenciar das variáveis.

Ex:
```
typedef struct {
      char nome[40] ;
      char  curso[30] ;
      int notas[5] ;
} ALUNO;
ALUNO  turma[20];
```

```
/* cap5_2.c */
#define MAX_ALUNOS  3

typedef struct {
                  char nome[40];
                  int nota;
                  } ALUNO;
main()
{
  ALUNO turma[MAX_ALUNOS];
  int i, total;
  float media;

  for(i=0; i< MAX_ALUNOS; i++)
  {
      printf("\n\nDigite o nome do %do. aluno : ", i+1);
      gets(turma[i].nome);

      printf("\nDigite a nota : ");
      scanf("%d%*c",&turma[i].nota);
  }

  total = 0;
  printf("\n ---------------------- \n");

  for(i=0; i< MAX_ALUNOS; i++)
  {
    printf("\nAluno : %8s \t Nota: %d",turma[i].nome,turma[i].nota);
    total += turma[i].nota;
  }
  media = (float) total / MAX_ALUNOS;

  printf("\n\n A media das notas e : %.2f", media);
}
```

Capítulo 6 : Ponteiros

Um ponteiro é um tipo de variável que contém um endereço de memória que é a localização de uma outra variável. Para definir uma variável como sendo um ponteiro deve-se colocar o símbolo '*' antes do nome da variável.

Ex: char * pont1;
int * pont2;
float * pont3;

Há dois operadores especiais de ponteiros: '*' e '&'. O '&' retorna o endereço de memória da variável, enquanto o '*' devolve o valor que está guardado no endereço que o ponteiro contém. Para podermos entender os ponteiros, devemos ter uma idéia da forma que as variáveis são armazenadas na memória.

Ex:

Endereço de memória	Nome da variável	Conteúdo
100	x	5
101		
102	pont	100
103		

Considerando 'x' uma variável do tipo inteiro e 'pont' um ponteiro para um inteiro, temos os seguintes resultados:

- x *retorna* 5 ;
- & x *retorna* 100;
- pont *retorna* 100;
- *pont *retorna* 5;

```
/* cap6_1.c */

main()
{
  int x;
  int *pont;

  printf("\nEstudo dos ponteiros\n");

  printf("\nDigite um valor para X: ");
  scanf("%d", &x);

  pont = &x;  /* o ponteiro recebe o endereco de x*/
```

```
  printf("\nx = %d",x);
  printf("\nEndereco de memoria de X = %p", &x);
  printf("\nConteudo de pont = %p", pont);
  printf("\nValor que esta no endereco apontado por pont = %d",*pont);

  /*alterar o conteudo de x atraves do ponteiro */
  *pont += 10;

  printf("\nX alterado pelo ponteiro = %d", x);

}
```

É possível copiar o conteúdo de um ponteiro para outro usando o operador de atribuição '='.
Ex:

```
int  x;
int  *p1,  *p2;
x = 10;
p1 = &x ;
p2 = p1;
printf("%d", *p2);  /* imprimirá 10 */
```

Só podemos usar duas operações aritméticas com ponteiros : + e - . Se você somar 1 ao ponteiro ele passará a apontar para a próxima posição de memória de acordo com o seu tipo. Por exemplo, se 'pont' é um ponteiro apontando para um **float** (tamanho de 4 bytes) e tem o conteúdo de 1000, a operação 'pont++' fará o conteúdo de 'pont' mudar para 1004 que é a posição onde está o próximo **float**.

Os ponteiros estão diretamente relacionados às matrizes. O nome de uma matriz sem um índice é o endereço do primeiro elemento e a indexação funciona como uma soma ao ponteiro. Por exemplo, para acessar o terceiro elemento de uma matriz, você poderia escrever :

mat[2] ou *(mat + 2)

Sempre que se declarar um ponteiro e não atribuir logo um valor, é aconselhável que seja atribuído NULL a ele. NULL é uma constante definida na biblioteca do C para representar os ponteiros nulos. Isto ajuda na verificação da validade dos ponteiros.
Ex: int *pont = NULL ;

Os ponteiros são muito usados para trabalhar com alocação dinâmica de memória em C. Este método de alocação é usada durante a execução do programa e permite um controle mais eficiente da utilização da memória pelo programa. As duas funções principais de alocação dinâmica são **malloc( )** e **free( )**. A função malloc( ) recebe como parâmetro o número de bytes que devem ser alocados e retorna um ponteiro sem tipo, isto significa que é preciso usar um *cast* para definir o tipo de ponteiro. O exemplo a seguir aloca 100 bytes de memória.

Ex:

```
char *p ;
p = ( char * )  malloc( 100 );
```

Quando a memória usada não for mais necessária, é preciso liberá-la para o sistema usando a função **free( )** que recebe um ponteiro como parâmetro.

Ex: `free( p );`

```
/* cap6_2.c */

main()
{
  char *p;
  int i;

  p = (char *) malloc(50);

  printf("\nDigite seu nome em minusculo : ");
  gets(p);
  printf("\n Bem vindo %s", p);
  printf("\n Seu nome em maiusculo : ");
  for(i=0; p[i]; i++) printf("%c",toupper(*(p+i)));

  free(p);
}
```

É possível também definir um ponteiro para uma estrutura. Considerando que há um gabarito de estrutura chamado 'aluno', o exemplo a seguir define um ponteiro para essa estrutura:

struct aluno * aluno_pont ;

Para acessar os membros de uma estrutura a partir de um ponteiro, utilizamos uma '->' ao invés do ponto '.' .

Ex: aluno_pont->nome;

aluno_pont->nota;

```
/* cap6_3.c */

struct pessoa {
            char nome[40];
            int idade;
            };

main()
{
   struct pessoa pes1, pes2, pes3;
   struct pessoa *pes_pont;
   int resp;

   strcpy(pes1.nome, "Marcos");
   pes1.idade = 23;

   strcpy(pes2.nome, "Diana");
   pes2.idade = 21;

   strcpy(pes3.nome, "Renaldo");
   pes3.idade = 28;

   printf("\nDigite um numero de 1 a 3 : ");
   scanf("%d", &resp);

   switch(resp)
   {
            case 1: pes_pont = &pes1; break;
            case 2: pes_pont = &pes2; break;
            case 3: pes_pont = &pes3; break;
            default:printf("\n opcao invalida");
                        exit(0);
   }

   printf("\nNome = %s ", pes_pont->nome);
   printf("\nIdade = %d",pes_pont->idade);

}
```

Capítulo 7: Funções

As funções são os blocos de construção de um programa em C. Cada função tem um nome e contém diversos comandos C. Elas são usadas para dividir um programa grande em diversas partes menores.

Ex:

```
main( )
{
      teste( );
}
teste( )
{
     printf("Funciona");
}
```

No exemplo acima é definida uma função chamada **teste( )**. Dentro do **main( )** simplesmente é feita uma chamada a esta função. As funções também aceitam argumentos, que são valores passados à função.

Ex:

```
main( )
{
      int valor;
      printf("Digite um numero : ");
      scanf("%d", &valor);
      cubo( valor );
}
cubo( int x )
{
      printf("%d ao cubo = %d", x, x * x * x);
}
```

As funções também podem retornar valores usando o comando **return**.

Ex:

```
main( )
{
      int   resultado;
      resultado  =  soma( 5, 3 );
      printf("%d", resultado);
}
soma( int x, int y)
{
      return x + y ;
}
```

```
/* cap7_1.c */

main()
{
  int x, y;
  int resultado;

  printf("\n POTENCIA\n");
  printf("\nDigite a base : ");
  scanf("%d", &x);
  printf("\nDigite o expoente : ");
  scanf("%d", &y);

  resultado = potencia(x, y);

  printf("\n%d elevado a %d = %d", x, y, resultado);

}

potencia(int base, int exp)
{
  int total =1;

  while(exp > 0)
  {
      total *= base;
      exp--;
  }
  return total;
}
```

As variáveis definidas dentro de uma função são chamadas de variáveis locais. Elas são criadas quando inicia o processamento da função onde elas estão e são destruídas quando a função termina, dessa forma as variáveis locais não guardam valores entre as execuções da função. Uma variável local não pode ser acessada fora da função, isto ajuda na divisão do programa em blocos. As variáveis que são declaradas fora de qualquer função (geralmente no início do programa) são chamadas de variáveis globais e podem ser acessadas em qualquer parte do programa. Há um modificador chamado **static** que se for aplicado às variáveis locais, faz com que elas guardem os seus valores mesmo após o encerramento da função.

```
/* cap7_2.c */

int global = 0;

main()
{
  int i, c1, c2;

      printf("\nCONT1 --- CONT2 --- GLOBAL \n");

  for(i=0; i<5; i++)
  {
      c1 = cont1();
      c2 = cont2();
      printf("\n  %d   ---   %d   ---  %d \n",c1, c2, global);
  }
}

cont1()
{
  int x=0;
  x++;
  global++;
  return x;
}

cont2()
{
  static int x=0;
  x++;
  global++;
  return x;
}
```

Há duas formas de passar argumentos para funções, por valor e por referência. O primeiro método copia o valor de um argumento para a variável de parâmetro da função, de forma que qualquer alteração que forem feitas nos parâmetros dentro da função não afetarão a variável original que foi passada à função. O segundo método copia o endereço de um argumento para a variável de parâmetro, isto significa que as alterações que forem feitas nos parâmetros dentro da função afetarão a variável que foi usada quando a função foi chamada. Este método de referência é implementada usando ponteiros.

```
/* cap7_3.c */

main()
{
  int x =0;

  por_valor(x);
  printf("\nChamada por valor. x = %d", x );

  por_referencia(&x);
  printf("\nChamada por referencia. x = %d", x );

}
```

```
por_valor(int a)
{
      a = 10;
}

por_referencia( int * a)
{
  *a = 10;
}
```

A forma de passagem de argumentos por referência é muito utilizado com matrizes e estruturas, lembrando que o nome de uma matriz sem o índice é um ponteiro para o 1º elemento da matriz. A vantagem de usar a passagem por referência com estruturas é que apenas um endereço é passado à função, ao invés de copiar toda a estrutura na memória.

```
/* cap7_4.c */

struct pessoa {
            char nome[40];
            char tel[10];
            char end[40];
            };

main()
{
  struct pessoa p1;

  strcpy(p1.nome, "marcos romero");
  strcpy(p1.tel, "226-7176");
  strcpy(p1.end, "Av. Dr. Freitas 2256");
  maiusculo(p1.nome);
  imprime_estru(&p1);
}

maiusculo(char *texto)
{
  int i;
  for(i=0; texto[i]; i++)
  {
      texto[i] = toupper(texto[i]);
  }
}

imprime_estru(struct pessoa *p)
{
  printf("\nNome = %s", p->nome);
  printf("\nTel = %s", p->tel);
  printf("\nEnd = %s", p->end);
}
```

O valor padrão de retorno de uma função é **int** ou nenhum. Se uma funçào devolve um outro tipo (como **float**) é preciso deixar isto explícito para o programa através do uso de protótipos de função que especificam o tipo de retorno e os tipos dos parâmteros. Esses protótipos são inseridos logo no começo do programa.

Ex:

```
/* prototipo */
float divisao(float x, float y);

main( )
{
        float a=5, b=2, res;
        res = divisao(a, b);
        printf("Resultado = %f", res);
}

float divisao(float x, float y)
{
        return x / y;
}
```

Capítulo 8 : Arquivos

Os arquivos são usados para guardar dados de forma permanente. Há dois tipos de arquivos: em modo texto e em modo binário.

Os arquivos em modo texto são organizados como seqüência de caracteres em linhas que são encerradas por caracteres '\n'. Neste formato podem haver códigos de formatação fazendo com que o número de caracteres lidos não sejam iguais àquele encontrado no arquivo.

Os arquivos em modo binário é uma seqüência de bytes que tem correspondência de um-para-um com os caracteres lidos, pois ele não possui nenhum tipo de código de formatação.

Para usarmos as funções que lidam com os arquivos, precisamos inserir o arquivo cabeçalho "stdio.h" no começo do programa usando o comando **#include**, pois ele contém informações sobre as estruturas e funções de arquivos.

Ex: #include "stdio.h"

Para manipular um arquivo é preciso usar um ponteiro de arquivo que é uma variável de ponteiro do tipo FILE definida em "stdio.h".

Ex: FILE *fp;

A função utilizada para abrir um arquivo é a "fopen( )" que recebe como argumentos o nome do arquivo, que pode ter a especificação de diretório, e o modo como ele será aberto. A tabela a seguir mostra os modos mais comuns.

modo texto	modo binário	operação
"r"	"rb"	Abre arquivo para leitura
"w"	"wb"	Abre arquivo para gravação
"a"	"ab"	Anexa à um arquivo já existente

Ex: fp = fopen("teste.txt", "w");

Para gravar caracteres em um arquivo aberto é utilizada a função **putc( )**.

Ex: putc('m' , fp);

Para ler caracteres de um arquivo aberto é utilizada a função **getc( )**.

Ex: char letra;
letra = getc(fp);

É muito importante fechar o arquivo assim que a sua operação seja finalizada. A função fclose( ) executa esta tarefa.

Ex: fclose(fp);

Há uma outra função utilitária chamada **feof( )** que detecta se o fim do arquivo foi alcançado.
Ex: if (feof(fp)) printf("fim de arquivo ");

```
/* cap8_1.c */

#include "stdio.h"

main()
{
  FILE *fp;
  int escolha, i;
  char texto[50];
  printf("\n Arquivo em modo texto \n\n\n");
  printf("\n1. Gravar Arquivo");
  printf("\n2. Ler Arquivo");
  printf("\n Opcao : ");
  scanf("%d%*c",&escolha);

  if(escolha == 1)
  {
      fp = fopen("teste.txt","w");
      printf("\nDigite um nome : ");
      gets(texto);
      for(i=0; texto[i]; i++) putc(texto[i], fp);

      fclose(fp);
      printf("\nSalvo em teste.txt");
  }
  else
  if(escolha == 2)
  {
   fp = fopen("teste.txt","r");
   if (fp == NULL) printf("\nErro ao abrir arquivo");
   else
   {
       printf("\n Conteudo do arquivo : \n");
       while(!feof(fp)) printf("%c", getc(fp) );
   }
  }
  else
  {
      printf("\nOpcao invalida.");
  }

}
```

As funções **getw( )** e **putw( )** trabalham com valores inteiros e funcionam da mesma forma que **getc( )** e **getw( )**. A função **fputs( )** escreve strings em um arquivo enquanto que a **fgets( )** lê strings a partir de um arquivo. A função **fgets( )** recebe como argumento o comprimento da string que deve ser lida, ela concluirá quando atingir este comprimento ou quando encontrar um '\n'.

```
Ex:   fputs("texto", fp);
      char nome[40];
      fgets(nome, 40, fp);
```

Há também as versões de printf( ) / scanf( ) que trabalham com arquivos, seus nomes são **fprintf( )** e **fscanf( )**.

```
Ex:   fprintf(fp, "%d", valor);
      fscanf(fp, "%d", &valor);
```

As funções **fread( )** e **fwrite( )** trabalham com blocos de dados e são as mais utilizadas quando se manipula muitos dados.

```
Ex:   int salarios[50];
      /* preencher salarios e abrir arquivos p/ escrita...  */
      fwrite(salarios, sizeof(salarios), 1, fp);
```

O comando **sizeof( )** retorna a quantidade de bytes que um tipo contém. Para ler de volta o bloco de dados use o **fread( ).**

```
Ex:   /* abrir arquivo p/ leitura... */
      fread(salarios, sizeof(salarios), 1, fp);
```

Há uma função que retorna a posição do arquivo para o início, esta função é a **rewind( )**, para usá-la, basta passar o ponteiro para o arquivo. A função **remove( )** apaga um arquivo, ela recebe como parâmetro o nome do arquivo (que pode ter os diretórios).

```
Ex:   remove("teste.txt");
```

```
/* cap8_2.c */
#include "stdio.h"

struct heroi {
            char nome[40];
            int energia;
            int moedas;
            };
main()
{
  int escolha;
  struct heroi h1;
  FILE *fp;

  printf("\n Arquivo em modo binario \n\n\n");
  printf("\n1. Gravar Arquivo");
  printf("\n2. Ler Arquivo");
  printf("\n Opcao : ");
  scanf("%d%*c",&escolha);

  switch(escolha)
  {
      case 1:     /* preencher informacoes do heroi */
                  strcpy(h1.nome, "Mestre");
                  h1.energia = 100;
                  h1.moedas = 15;

                  /* salvar em arquivo */
                  fp = fopen("heroi.txt","wb");
                  if(fp == NULL)
                  {
                        printf("\n erro ao abrir arquivo");
                        exit(0);
                  }
                  else
                  {
                     fwrite(&h1, sizeof(struct heroi), 1, fp);
                     fclose(fp);
                     printf("\n Salvo em heroi.txt");
                  }
                  break;

      case 2:     fp = fopen("heroi.txt","rb");
                  if(fp == NULL)
                  {
                        printf("\n erro ao abrir arquivo");
                        exit(0);
                  }else
                  {
                        fread(&h1, sizeof(struct heroi), 1, fp);
                        fclose(fp);
                        printf("\n Dados do Heroi : \n");
                        printf("\n Nome : %s", h1.nome);
                        printf("\n Energia : %d", h1.energia);
                        printf("\n Moedas : %d", h1.moedas);
                  }
                  break;

      default:  printf("\nOpcao invalida");
  }
}
```

- Programa de Exemplo da Parte II

O programa seguinte mostra o desenvolvimento de uma agenda de endereços bem simples. Os dados são armazenadas em uma matriz de estrutura e há a possibilidade de gravar e recuperar as informações em arquivo. As diversas operações do programa são divididos em funções para facilitar a codificação.

```
/*
Nome    : agenda.c
Autor   : Marcos Romero

      Programa de exemplo da parte II
*/

#include "stdio.h"

/* valores constantes */
#define MAX_ITENS 100

typedef struct {
            char nome[30];
            char telefone[15];
            char endereco[30];
      } INFO;

INFO agenda[MAX_ITENS];

int n_itens = 0;

main()
{
  int escolha;

  printf("\n\n *** EXEMPLO DE AGENDA *** \n");

  while(1)
  {
      printf("\n\n M E N U \n");
      printf("\n 1. Entrar dados");
      printf("\n 2. Listar dados");
      printf("\n 3. Salvar");
      printf("\n 4. Carregar");
      printf("\n 5. Encerrar programa");

      printf("\n\n Escolha: ");
      scanf("%d%*c", &escolha);

      switch(escolha)
      {
        case 1:
                  Entrar_Dados();
                  break;

        case 2:
                  Listar_Dados();
                  break;
```

```
        case 3:
                  Salvar();
                  break;

        case 4:
                  Carregar();
                  break;

        case 5:
                  exit(0);
                  break;

        default: printf("\n Opcao invalida\n ");
                   break;
      }/*fim switch*/
  } /*fim while*/
}/*fim main*/

Entrar_Dados()
{
  char resp;

  do {

      if(n_itens >= MAX_ITENS)
      {
            printf("\nAgenda cheia");
            return;
      }

      printf("\n Nome : ");
      gets(agenda[n_itens].nome);
      printf("\n Telefone : ");
      gets(agenda[n_itens].telefone);
      printf("\n Endereco : ");
      gets(agenda[n_itens].endereco);

      n_itens++;

      printf("\n Continuar(s/n) : ");
      scanf("%c%*c", &resp);

  } while(resp == 's' || resp == 'S');
}

Listar_Dados()
{
      int i;

      if(n_itens == 0)
      {
        printf("\n Agenda vazia.");
        return;
      }
      for(i=0; i < n_itens; i++)
      {
            printf("\n %s --- %s --- %s", agenda[i].nome,
                  agenda[i].telefone, agenda[i].endereco);
      }
}
```

```
Salvar()
{
  FILE *fp;

      if(n_itens == 0)
      {
        printf("\n Agenda vazia.");
        return;
      }

  fp = fopen("dados.bd", "wb");

  if(fp==NULL)
  {
      printf("\n Erro ao abrir arquivo.");
      return;
  }
  //salvar o numero de itens primeiro
  putw(n_itens, fp);

  //salvar todos os dados em um unico bloco
  fwrite(agenda, sizeof(INFO), n_itens, fp);

  fclose(fp);

  printf("\n Salvo em dados.bd");
}

Carregar()
{
  FILE *fp;

  fp = fopen("dados.bd", "rb");

  if(fp==NULL)
  {
      printf("\n Erro ao abrir arquivo.");
      return;
  }
  //ler o numero de itens primeiro
  n_itens = getw(fp);

  //recuperar todos os dados em um unico bloco
  fread(agenda, sizeof(INFO), n_itens, fp);

  fclose(fp);

  printf("\n Dados recuperados.");

}
```

Parte III : Programação Orientada a Objetos com C++

Capítulo 9 : Primeiros programas em C++

A linguagem C++ foi criada com o objetivo de incorporar à linguagem C os conceitos de programação orientada a objetos. Além disso, ela deveria ser totalmente compatível com programas em C, e ser tão rápido quanto a linguagem C.

A linguagem C++ é considerada híbrida porque permite a programação de forma estruturada (como é feito em C) e a programação orientada a objetos. Neste capítulo veremos diversos novos tópicos de C++ mas que não estão diretamente relacionados a orientação a objetos.

```
// cap9_1.cpp

#include "iostream.h"

const int MAX_NUM = 3;

void main()
{
  cout << "\n Bem vindo ao C++ \n";
  cout << "Digite " << MAX_NUM << " numeros : ";

  int numeros[MAX_NUM];

  for(int i=0; i < MAX_NUM; i++)
  {
      cin >> numeros[i];
  }

  cout << "\n Os quadrados dos numeros : \n";
  for(i=0; i < MAX_NUM; i++)
  {
   cout << numeros[i] << " --- " << numeros[i] * numeros[i] << endl;
  }
}
```

Na primeira linha é usada uma forma de comentário, tudo que vier depois de duas barras ´//´ até o final da linha é considerado comentário. Em seguida é incluído o arquivo de cabeçalho “iostream.h” que é necessário para as novas funções de entrada e saída. Pode-se definir valores constantes usando o comando “**const**” antes da definição da variável. O C++ requer que sejam especificados os parâmetros e valor de retorno de uma função, por isso a função main( ) está declarada como **void main( ),** sendo que **void** indica que não existe. É permitida a definição de variáveis em qualquer posição da função. O C++ usa um novo mecanismo de entrada e saída de informações, que são os objetos **cout** e **cin.**

```
// cap9_2.cpp

#include "iostream.h"

//prototipos das funcoes

void inicia_var(int &var, int valor = 0);
void inicia_var(float &var, float valor = 0.0);

void main()
{
   int var_inteira;
   float var_fracionaria;

   inicia_var(var_inteira, 5);
   cout << "\n var_inteira = " << var_inteira;

   inicia_var(var_inteira);
   cout << "\n var_inteira = " << var_inteira;

   inicia_var(var_fracionaria, 3.75);
   cout << "\n var_fracionaria = " << var_fracionaria;

}

void inicia_var(int & var, int valor)
{
   var = valor;
}

void inicia_var(float & var, float valor)
{
   var = valor;
}
```

C++ fornece novas características às funções. Os protótipos das funções passaram a ser obrigatórias para que o C++ possa se certificar de que elas estão sendo chamadas de forma correta. É possível passar uma variável por referência (usando o ´&´ na declaração do parâmetro) fazendo com que a variável original sofra as alterações feitas dentro da função. Há também o conceito de sobrecarga de função que permite que várias funções tenham o mesmo nome, desde que tenham argumentos de tipos diferentes ou números diferentes de argumentos. Finalmente, há a opção de atribuir valores padrões à argumentos das funções, estes valores devem ser especificados no protótipo da função.

Capítulo 10: Classes e Objetos

A programação orientada a objetos é baseada no conceito de classes. Uma classe descreve os atributos e ações que os objetos possuem. Por exemplo, poderíamos ter uma classe chamada "Pessoa" e diversos objetos desta classe como "Marcos, "Roberto". A definição de uma classe é muito semelhante à definição de uma estrutura, porém é possível definir funções dentro da classe.

```
// cap10_1.cpp

#include "iostream.h"

class retangulo {
            public :
                  int altura;
                  int largura;

                  int area()
                  {
                    return altura * largura;
                  }
            };

void main()
{
   retangulo r1, r2;

   r1.altura = 5;
   r1.largura = 3;

   r2.altura = 8;
   r2.largura = 4;

   cout << "\n Area do retangulo 1 = " << r1.area();
   cout << "\n Area do retangulo 2 = " << r2.area();

}
```

O C++ permite que as funções das classes sejam definidas fora da declaração da classe, bastando colocar um protótipo da função na classe. A vantagem disto é que fica fácil de visualizar todas as funções que pertence à classe.

É possível acessar as funções da classe a partir de ponteiros da classe usando o operador ´->´, da mesma forma que um ponteiro de estrutura acessa os dados da estrutura.

As classes têm uma função especial chamada de **Construtor** que é usado para iniciar seus dados. O construtor é chamado automaticamente quando um objeto é criado.

```
// cap10_2.cpp

#include "iostream.h"

class relogio {
            public:
                  relogio();
                  void ajusta_hora(int h, int m=0, int s=0);
                  void mostra_hora();

                  int hora;
                  int minuto;
                  int segundo;
      };

relogio::relogio()
{
  hora = 12;
  minuto = 0;
  segundo = 0;
}

void relogio::ajusta_hora(int h, int m, int s)
{
  if(h >= 0 && h < 24) hora = h;
  if(m >=0 && m < 60) minuto = m;
  if(s >=0 && s < 60) segundo = s;
}

void relogio::mostra_hora()
{
      cout << hora << " horas, "
             << minuto << " minutos, "
             << segundo << " segundos.\n";
}

void main()
{
  relogio r1;
  relogio * r_ptr; //ponteiro

  cout << "\nHora definida no Construtor :";
  r1.mostra_hora();

  r1.ajusta_hora(8, 45, 30);

  cout << "\nHora no r1 :";
  r1.mostra_hora();

  r_ptr = &r1; //o ponteiro recebe o endereco de r1
  r_ptr->ajusta_hora(15, 30);

  cout << "\nHora no r_ptr : ";
  r_ptr->mostra_hora();
  cout << "\nHora no r1    : ";
  r1.mostra_hora();
}
```

Os membros de uma classe podem ser especificados como protegidos (**protected**), fazendo com que eles não possam ser acessados fora da classe. O objetivo disso é limitar a interface da classe e tornar a classe responsável por controlar o uso de seus dados externos. Pode-se especificar também o nível de acesso como privativo (**private**). A diferença entre protected e private se tornará clara no capítulo sobre herança.

Eventualmente, pode ocorrer que uma função que não faz parte de uma classe precise acessar os membros protegidos desta classe. É possível fazer isso declarando na classe que a função é amiga (**friend**).

```
// cap10_3.cpp

#include "iostream.h"

class empregado {
            friend void aumentar_salario(empregado & e, int valor);

            public:
                  empregado()
                  {
                        salario = 100;
                  }
                  int vlr_salario()
                  {
                     return salario;
                  }
            protected:
                  int salario;
};

void aumentar_salario(empregado & e, int valor)
{
  //o acesso a variavel salario e permitido aqui
  //por que esta funcao e friend da classe empregado

  e.salario += valor;
}

void main()
{
  empregado emp;

  //a linha seguinte iria gerar um erro :
  // emp.salario = 1500;

  cout << "\n Salario = " << emp.vlr_salario();
  aumentar_salario(emp, 200);
  cout << "\n Novo salario = " << emp.vlr_salario();

}
```

Capítulo 11 : Melhorando as classes

O construtor de uma classe é um tipo especial de função, sendo assim ele pode receber argumentos, pode ser sobrecarregado e ter valores padrões. Os argumentos são passados no momento da criação da variável.

Pode-se utilizar o comando **static** antes da definição de uma variável na classe. Isto faz com que o conteúdo desta variável seja comum a todos os objetos, ou seja, ela não pertence a nenhum objeto, mas sim a classe.

Da mesma forma que existe o construtor, há outra função especial conhecido como destruidores que são chamados automaticamente quando os objetos são finalizados. O nome do destrutor é o nome da classe com o símbolo de negação ´~´ na frente.

```
// cap11_1.cpp

#include "iostream.h"
#include "string.h"


class Aluno {
      public :
            //construtor
            Aluno(char * p_nome = "Sem nome")
            {
               strcpy(nome, p_nome);
               n_alunos++;
            }
            //destruidor
            ~Aluno()
            {
              n_alunos--;
            }
            int num_alunos()
            {
              return n_alunos;
            }
            char nome[40];
      protected:
            static int n_alunos;
};

//alocar memoria p/ membro static
int Aluno::n_alunos = 0;
```

```
void fn()
{
  Aluno a4("Diana");

  cout << "\n Aluna :" << a4.nome;
  cout << "\nNumero de alunos = " << a4.num_alunos();
}

void main()
{
  Aluno a1;
  Aluno a2("Marcos");

  cout << "\n Criado alunos : " << a1.nome << " e " << a2.nome;
  cout << "\n Numero de alunos = "<< a1.num_alunos();

  Aluno a3("Andre");

  cout << "\n Criado o " << a3.nome;
  cout << "\n Numero de alunos = "<< a1.num_alunos();

  fn();

  cout << "\n Numero de alunos = "<< a1.num_alunos();

}
```

O C++ fornece novos mecanismo para trabalhar com alocação de memória: **new** para alocar a memória e **delete** para liberá-la.

Ex:
Aluno * p_aluno;
p_aluno = new Aluno;
delete p_aluno;

É possível alocar matrizes da seguinte forma:

Aluno *p_aluno;
p_aluno = new aluno[10]; // aloca memória p/ 10 alunos
delete [] p_aluno; // libera a memória de toda a matriz

Deve-se ter um cuidado especial com classes que trabalham com alocação dinâmica de memória. O código para liberação de memória deve ser posto no **destruidor** da classe. Sempre que for preciso passar um objeto para uma função, passe-o por referência, e evite usar o sinal de atribuição ´=´ entre classes.

```
// cap11_2.cpp

#include "iostream.h"
#include "string.h"


class Cliente {

            public :
                  char * nome;

                  //construtor
                  Cliente( char * _nome = "sem nome")
                  {
                    nome = new char[ strlen(_nome) + 1 ];
                    strcpy(nome, _nome);
                  }

                  //destruidor
                  ~Cliente()
                  {
                    delete[] nome;
                  }
      };

void main()
{
   Cliente c1("Jonas"), c2("Mateus");

   cout << "\nCliente 1 : " << c1.nome;
   cout << "\nCliente 2 : " << c2.nome;

}
```

Capítulo 12 : Herança e Polimorfismo

- Herança:

A herança é um mecanismo que permite que uma nova classe herde os atributos de uma classe já existente. Este é um conceito fundamental da orientação a objetos e está diretamente ligado à reutilização de software.

Ex:

```
class veiculo
{
        //definição...
};

class carro: public veiculo
{
        //definição das novas características de carro...
};
class moto: public veiculo
{
        //definição das novas características de moto...
};
```

A classe **veiculo** é chamada de classe base, ela possui somente informações e funções que são comuns a todos os tipos de veículos, por exemplo, "peso", "mover( )". A classe **carro** herda as propriedades da classe veiculo, ou seja, os atributos da classe carro são aqueles que estão definidos dentro da classe carro mais os que estão definidos na classe veiculo. Podemos dizer também que um carro É UM veículo. Da mesma forma, a classe **moto** herda da classe veículos, então uma moto É UM veículo.

Esta relação é muito importante. Suponhamos que temos uma função que receba um "veículo" como argumento, poderíamos então passar variáveis da classe veículo, da classe carro, da classe moto, ou de qualquer classe que herde de veículo.

Ex:

```
void fn( veiculo & v )
{
      // processamento...
}
void main( )
{
      veiculo  v1;
      carro  ferrari;
      moto  suzuki;

      fn( v1 );
      fn( ferrari );
      fn( suzuki );
}
```

Há um outro tipo de relação entre classes chamado de **Composição**, que não deve ser confundido com a Herança. A expressão que está ligada à Composição é a "TEM UM", por exemplo, um carro TEM UM motor.

Ex:
```
class carro: public veiculo
{
      motor m;
}
```

```
// cap12_1.cpp

#include "iostream.h"

class Ponto {
      public:
            //construtor
            Ponto(int _x = 0, int _y =0)
            {
              x = _x;
              y = _y;
            }
            void def_ponto(int _x, int _y)
            {
              x = _x;
              y = _y;
            }
            int vlr_x()
            {
              return x;
            }
            int vlr_y()
            {
              return y;
            }

            void imprime()
            {
              cout <<"\n PONTO :";
              cout <<" [ " << x << "," << y <<" ]\n";
            }
      protected:
            int x, y;
};
class Reta : public Ponto
{
      public:
            //construtor
            Reta(int xf=0, int yf=0, int xi=0, int yi=0): Ponto(xi, yi)
            {
              x_final = xf;
              y_final = yf;
            }

            void def_final(int xf, int yf)
            {
              x_final = xf;
              y_final = yf;
            }
```

```
            int vlr_x_final()
            {
              return x_final;
            }
            int vlr_y_final()
            {
              return y_final;
            }
            void imprime()
            {
              cout << "\n RETA : ";
              cout <<" [ " << x << "," << y <<" ] --- [ "
                     << x_final << "," << y_final <<" ]\n";
            }
      protected:
            int x_final, y_final;
};
void main()
{
  Ponto p1, p2(5,2);
  Reta r1, r2(8,4), r3(10, 6, -2, 1);

  cout << "\nCoordenadas do Ponto 1 :";
  p1.imprime();
  cout << "\nCoordenadas do Ponto 2 :";
  p2.imprime();
  p1.def_ponto(-3, 6);
  cout << "\nNovas coordenadas do Ponto 1 :";
  p1.imprime();

  cout <<"\n Coordenadas da Reta 1 : ";
  r1.imprime();
  cout <<"\n Coordenadas da Reta 2 : ";
  r2.imprime();
  cout <<"\n Coordenadas da Reta 3 : ";
  r3.imprime();

  r1.def_ponto(-2, -4);
  r1.def_final(7, 3);
  cout << "\n Novas coordenadas da Reta 1 :";
  r1.imprime();
}
```

- Polimorfismo

Levando em conta o programa “cap12_1.cpp”, imaginemos a seguinte situação:

```
void fn( Ponto & p)
{
        p->imprime( );
}
void main( )
{
        Ponto p1;
        Reta r1;
        fn( p1 );    // sem problemas
        fn( r1 );    // Qual função “imprime( )” será executada ?
}                    // A da classe Ponto ou da classe Reta ?
```

Na forma que as classes foram definidas, a função chamada será sempre a da classe Ponto. Gostaríamos que o programa pudesse verificar qual o tipo real do objeto e chamasse a sua função correspondente. O mecanismo que implementa isso é chamado de ligação dinâmica ou polimorfismo. Para indicar que uma função terá Polimorfismo deve-se colocar a palavra "**virtual**" antes de sua declaração na classe base. É recomendado colocar a palavra "virtual" também nas subclasses, apesar de não ser necessário.

Ex:
```
class  Ponto {
        // definições...
        virtual void imprime( );
        };
```

Na programação orientada a objetos há o conceito de **classes abstratas** que são classes que não possui todas as suas funções implementadas. Elas existem com o objetivo de servirem como base para uma hierarquia de classes. As subclasses é que vão implementar as funções que faltam da classe abstrata. Não é possível definir objetos de uma classe abstrata, pois a sua definição ficaria incompleta, mas é possível definir ponteiros ou referências. Para indicar que uma função virtual não possui implementação coloca-se "=0" depois de sua declaração.

Ex: virtual void Imprime( ) = **0**;

Dica: Se uma classe possuir um destruidor e pertencer à uma hierarquia de classes, é aconselhável que a classe base tenha um destruidor declarado como virtual.

```
// cap12_2.cpp

#include "iostream.h"


//classe abstrata
class Forma2d {
            public:
                  virtual float area()
                  {
                        return 0.0;
                  }
                  virtual void mostra_nome() = 0;
};
```

```
class Retangulo : public Forma2d
{
      public :
            //construtor
            Retangulo(float larg = 0.0, float alt = 0.0)
            {
              largura = larg;
              altura = alt;
            }
            virtual float area()
            {
              return largura * altura;
            }

            virtual void mostra_nome()
            {
              cout << "\n Retangulo ";
            }
      protected:
            float largura, altura;
};

class Circulo : public Forma2d
{
  public:
            Circulo( float r = 0.0)
            {
              raio = r;
            }
            virtual float area()
            {
              return 3.1416 * raio * raio;
            }
            virtual void mostra_nome()
            {
              cout << "\n Circulo ";
            }
  protected:
            float raio;
};

void main()
{
   Retangulo r1(5,3), r2(10, 8);
   Circulo c1(4);

   Forma2d * formas[3];

   formas[0] = &r1;
   formas[1] = &r2;
   formas[2] = &c1;

   //ajustar a saida para mostrar duas casas decimais
   cout.precision(2);

   for(int i=0; i < 3; i++)
   {
       formas[i]->mostra_nome();
       cout << "\n Area = " << formas[i]->area() << endl;
   }
}
```

- *Programa de exemplo da Parte III*

Este programa tem como objetivo controlar as contas de um banco. Foi usada uma abordagem orientada a objetos, de forma que existe uma classe abstrata chamada **conta** que descreve as características que são comuns a todo tipo de conta. Neste exemplo só há dois tipos de contas: Conta **Corrente** e **Poupança**. A diferença entre elas está na retirada de dinheiro, pois na conta do tipo poupança será cobrada uma taxa alta cada vez que for feito uma retirada, enquanto que na conta corrente só será cobrada uma pequena taxa caso o saldo esteja muito baixo.

```
//banco.cpp
//Programa exemplo da parte III
//Autor : Marcos Romero

#include "iostream.h"
#include "string.h"
#include "stdlib.h"
#include "stdio.h"

//prototipos
void Criar_Conta();
void Acessar_Conta();



class Conta
{
 public:

  Conta(int num, char * _cliente)
  {
       conta_num = num;
       strcpy(cliente, _cliente);
       saldo = 0.0;
       contador++; //acompanha quantas contas estao abertas
  }
  int Conta_Num()
  {
   return conta_num;
  }
  float Saldo()
  {
   return saldo;
  }
  static int Quant_Contas()
  {
   return contador;
  }
  void Deposito(float valor)
  {
   if(valor > 0.0) saldo += valor;
  }

  virtual void Retirada(float valor) = 0;
```

```
  void Exibir()
  {
   cout << "\nConta : " << conta_num
            << " --- "  << cliente
            << " --- Tipo : " << tipo;
  }
 protected:

  int conta_num;
  float saldo;
  char cliente[50];
  char tipo[20];

  static int contador;

};

//alocar variavel estatica
int Conta::contador = 0;



class Corrente : public Conta
{
 public:
  Corrente(int num, char * _cliente):Conta(num, _cliente)
  {
      strcpy(tipo, "Corrente");
  }

  // sobrecarregar funcao virtual pura
  virtual void Retirada(float valor)
  {
      if (saldo < valor)
      {
            cout << "\nInsuficiencia de fundos: saldo = " << saldo ;
      }
      else
      {
        saldo -= valor;
        // se saldo baixo, cobrar taxa de servico
        if (saldo < 500.00)
        {
         saldo -= 0.20;
        }
      }

  }

};
```

```
class Poupanca : public Conta
{
 public:
  Poupanca(int num, char * _cliente):Conta(num, _cliente)
  {
      strcpy(tipo, "Poupanca");
  }

  virtual void Retirada(float valor)
  {
       if (saldo < valor)
       {
        cout << "Insuficiencia de fundos: saldo = " << saldo;
       }
       else
       {
        saldo -= valor;
        saldo -= 5.00; //taxa de retirada da poupanca
      }
  }
};

const int MAX_CONTAS = 100;

Conta * conta_lista[MAX_CONTAS];

void main()
{
  int escolha;

  cout << "\n\n *** CONTROLE DE CONTAS EM UM BANCO *** \n";

  while(1)
  {
      cout << "\n\n M E N U \n"
             << "\n 1. Criar nova conta"
             << "\n 2. Acessar conta"
             << "\n 3. Encerrar programa"
             << "\n\n Escolha: ";
      cin >> escolha;

      switch(escolha)
      {
        case 1:
                  Criar_Conta();
                  break;

        case 2:
                  Acessar_Conta();
                  break;

        case 3:
                  //limpar memoria das contas
                  int n_contas = Conta::Quant_Contas();
                  for(int i=0; i < n_contas; i++)
                  {
                    delete conta_lista[i];
                  }

                  exit(0);
                  break;
```

```
        default: cout <<"\n Opcao invalida\n ";
                    break;
       }/*fim switch*/
  } /*fim while*/

}


void Criar_Conta()
{
  int novo_num;
  char novo_cliente[50];
  float dep_inicial;
  int novo_tipo;

  cout << "\n Criando nova conta : \n";

  cout << "\n Digite o numero da nova conta: ";
  cin >> novo_num;

  cout << "\n Digite o nome do cliente: ";
  gets(novo_cliente);

  do {
  cout << "\n 1. Corrente"
         << "\n 2. Poupanca"
         << "\n Digite o tipo da nova conta: ";
  cin >> novo_tipo;
  } while (novo_tipo != 1 && novo_tipo !=2);

  cout << "\n Digite o valor do deposito inicial: ";
  cin >> dep_inicial;

  int posicao = Conta::Quant_Contas();

  if(novo_tipo == 1)
  {
      conta_lista[posicao] = new Corrente(novo_num, novo_cliente);
      conta_lista[posicao]->Deposito(dep_inicial);
  }
  else
  {
      conta_lista[posicao] = new Poupanca(novo_num, novo_cliente);
      conta_lista[posicao]->Deposito(dep_inicial);
  }
  cout << "\nConta criada : \n";
  conta_lista[posicao]->Exibir();

}
```

```
void Acessar_Conta()
{
      int n_contas = Conta::Quant_Contas();
      if(n_contas == 0)
      {
        cout << "Nao existe contas";
        return;
      }
      cout << "\n\nContas Existentes : ";
      for(int i=0; i < n_contas; i++)
      {
        conta_lista[i]->Exibir();
      }

      int num, id=-1;
      do {
            cout << "\n Numero da conta : ";
            cin >> num;
            for(i=0; i < n_contas; i++)
            {
              if(num == conta_lista[i]->Conta_Num())
              {
                  id = i;
                  break;
              }
            }
            if(id == -1) cout << "\nNumero invalido.";

      } while (id == -1);

   conta_lista[i]->Exibir();
   cout.precision(2);
   cout << "\n Saldo = " << conta_lista[i]->Saldo();

   cout << "\n\n Transacoes : ";
   float vlr;
   do {
         cout << "\n\n Digite um valor positivo p/ deposito,\n"
                  << " negativo p/ retirada e 0 p/ sair.";
         cout << "\n Valor : " ;
         cin >> vlr;

         if(vlr > 0.0) conta_lista[i]->Deposito(vlr);
         else if(vlr < 0.0) conta_lista[i]->Retirada(-vlr);

         cout <<"\n Saldo atual : " << conta_lista[i]->Saldo();

   } while(vlr != 0.0);

}
```

Bibliografia:

- Schildt, Herbert. **Turbo C - Guia do Usuário**. São Paulo: McGraw-Hill, 1988.

- Davis, Stephen. **C++ para Leigos**. São Paulo: Berkeley, 1995.

- Deitel, Harley. **C++: Como Programar**. Porto Alegre: Bookman, 2000.

- LaMothe, André. **Tricks of the Windows Game Programming Gurus**. Indianapolis: SAMS, 1999.