RESUMO – LINGUAGEM C

História

A linguagem C foi criada por Dennis M. Ritchie e Ken Thompson em 1972, no laboratório Bell. Seu desenvolvimento começou com uma linguagem mais antiga, chamada BCPL, criada por Martin Richards. Esta linguagem influenciou o desenvolvimento da linguagem B, criada por Ken Thompson, e posteriormente levou ao desenvolvimento de C.

A versão de C fornecida com o sistema Unix versão 5 foi o padrão da linguagem por muitos anos. Um grande número de implementações, porém, foi criado, com a popularidade dos microcomputadores, e isso gerou certa incompatibilidade entre essas implementações. Foi baseado nesta situação que em 1983 o ANSI (American National Standards Institute) estabeleceu um comitê para criar um padrão para definir a linguagem C. Atualmente, todos os principais compiladores de C implementam o padrão C ANSI.

Características

- ***C é uma linguagem estruturada***

C utiliza funções para dividir o seu código em sub-rotinas, tornando o trabalho do programador mais eficaz. Você pode criar bibliotecas, que poderão ser compartilhadas por qualquer programa que utilize suas rotinas. Há também a possibilidade de dividir o seu código-fonte em vários arquivos, onde cada arquivo pode ter rotinas para uma função específica, tornando o desenvolvimento muito mais organizado.

- ***C é uma linguagem de médio nível***

Isto significa que C possui recursos de alto nível, e ao mesmo tempo permite a manipulação de bits, bytes e endereços - os elementos mais básicos para a funcionalidade de um computador.

- ***C é uma linguagem compilada***

Uma linguagem de computador pode ser compilada ou interpretada. C é compilada, isto é, o compilador da linguagem lê o arquivo do código-fonte, processa-o e cria um arquivo em formato binário (objeto). Para este arquivo se tornar um programa executável, entra em cena o linker, que "liga" o binário gerado na compilação do programa com as bibliotecas da linguagem.

O fato de C ser compilada torna esta linguagem à opção certa quando se quer velocidade. Linguagens interpretadas geralmente são bem mais lentas do que as compiladas.

COMPILADOR LINKER
(código-fonte) ----------> (código-objeto) ----------> (programa executável)

- ***C é uma linguagem portável***

Um programa feito em C pode ser compilado em outras plataformas e funcionar sem problemas. C foi desenvolvida para ser uma linguagem portável, e a padronização feita pelo ANSI melhorou ainda mais esta característica. É claro que se deve seguir certas práticas para tornar o código portável para outras plataformas, como evitar o uso de bibliotecas particulares a uma plataforma específica, usar diretivas do compilador quando necessário, adotar o padrão ANSI, etc.

Estrutura básica de um programa em C

Iremos começar estudando como é a estrutura básica de um programa bem simples em C, que escreverá na tela a mensagem "Meu primeiro programa!". A seguir você tem o código do programa. As linhas estão numeradas para facilitar o entendimento.

```
1  #include <stdio.h>
2
3   int main()
4  {
5                          printf("Meu Primeiro Programa!\n");
6  }
```

Agora vamos explicar linha por linha. A linha 1 contém a diretiva #include <stdio.h>, e serve para incluir as funções de uma biblioteca em seu programa. Neste caso, estamos incluindo a biblioteca de entrada e saída padrão (standard input/output), necessária para a utilização da função printf(). C é uma linguagem que diferencia minúsculas de MAIÚSCULAS (case sensitive). As palavras-chave e os comandos da linguagem devem ser escritos em minúsculo. Se você declarar variáveis, funções, tipos, etc, utilizando caracteres maiúsculos, não se esqueça de referenciá-los da forma correta.

A linha 2 não contém nada... Está em branco. Você pode adicionar linhas em branco, espaços e tabulações em seu código para torná-lo mais fácil de ser lido.

A linha 3 contém o início da função main(). Esta função marca o início da execução do programa e deve existir em algum lugar do código para este funcionar corretamente. Se o seu programa tiver somente uma função, ela deverá ser main(). Este int, antes do main(), está dizendo que main() deve retornar um valor para o sistema, indicando sucesso ou falha. Este valor é um número inteiro, por isso int.

Na linha 4 vemos o caractere abre-chaves, um delimitador de bloco, serve para abrir um bloco de código. O que estiver dentro do par de chaves pertence à função main(), neste caso.

Mas é na linha 5 que o nosso programa cumpre a sua crucial missão: escrever a mensagem "Meu Primeiro Programa!" na tela. O que você precisa saber agora é que a função printf() é a principal função para escrita no console (saída padrão. O "\n" é um caractere especial, e serve para pular para a próxima linha, assim que acabar de escrever a mensagem.

OBS: os comandos em C são terminados com um ";" (ponto-e-vírgula). Não esqueça.

E finalmente chegamos à linha 6, onde encontramos as fecha-chaves, que significa o fim do bloco de código.

Em C você tem a liberdade de escrever o seu código em qualquer estilo. Como exemplo o nosso programa:

```
#include <stdio.h>
        int main()    {     printf("Meu Primeiro Programa!\n"); }
```

O compilador aceitará sem problemas se você escrever seu código assim, tudo em uma linha. É claro que desta forma o código fica muito mais difícil de entender do que da outra forma, mas o compilador aceita. Também é uma prática saudável e recomendável deixar seu código fácil de ser lido e entendido utilizando espaços, tabulações e quebras de linha.

Bibliotecas

Uma biblioteca é uma coleção de funções. Quando seu programa referencia uma função contida em uma biblioteca, o linkeditor (responsável por ligar o código-fonte) procura essa função e adiciona seus códigos ao seu programa. Desta forma somente as funções que você realmente usa em seu programa são acrescentados ao arquivo executável.

Cada função definida na biblioteca C padrão tem um arquivo de cabeçalho associada a ela. Os arquivos de cabeçalho que relacionam as funções que você utiliza em seus programas devem ser incluídos (usando ***#include***) em seu programa.

Há duas razões para isto. Primeiro muitas funções da biblioteca padrão trabalham com seus próprios tipos de dados específicos, aos quais seu programa deve ter acesso. Esses tipos de dados são definidos em *arquivos de cabeçalho* fornecidos para cada função. Um dos exemplos mais comuns é o arquivo *STDIO.H*, que fornece o tipo FILE necessário para as operações com arquivos de disco. A segunda razão para incluir arquivos de cabeçalho é obter os protótipos da biblioteca padrão.

Se os arquivos de cabeçalho seguem o padrão C ANSI, eles também contêm protótipos completos para as funções relacionadas com o arquivo de cabeçalho. Isso fornece um método de verificação mais forte que aquele anteriormente disponível ao programador

C. Incluindo os arquivos de cabeçalho que correspondem às funções padrões utilizados pelo seu programa, você pode encontrar erros potenciais de inconsistências de tipos.

Na tabela abaixo algumas bibliotecas:

Arquivo de Cabeçalho	***Finalidades***
ASSERT.H	Define a macro assert()
CTYPE.H	Manipulação de caracteres
ERRNO.H	Apresentação de erros
FLOAT.H	Define valores em ponto flutuante dependentes da implementação
LIMITS.H	Define limites em ponto flutuante dependentes da implementação
LOCALE.H	Suporta localização
MATH.H	Diversas definições usadas pela biblioteca de matemática
SETJMP.H	Suporta desvios não-locais
SIGNAL.H	Suporta manipulação de sinais
STDARG.H	Suporta listas de argumentos de comprimento variável.
STDDEF.H	Define algumas constantes normalmente usadas
STDIO.H	Suporta e/s com arquivos
STDLIB.H	Declarações miscelâneas
STRING.H	Suporta funções de string
TIME.H	Suporta as funções de horário do sistema

Observe que o nome da biblioteca estará entre os sinais de menor e maior (<>). Se o nome da biblioteca for delimitado pelos sinais de maior e menor, o compilador C procurará o arquivo especificado no sue diretório de arquivos de cabeçalho. Se o arquivo for encontrado, será utilizado, se o compilador não o encontrar, ele irá procurar então no diretório atual, ou em um diretório que você especificar.

Se o nome da biblioteca for delimitado por aspas duplas (“”), o compilador procurará o arquivo no diretório atual. Se o nome da biblioteca for delimitado por aspas duplas significa que esta biblioteca foi criada pelo usuário.

Tipos, Variáveis e Constantes

- ***Os Tipos de Dados***

Em C temos 5 tipos básicos de dados, de onde todos os outros tipos são derivados. São eles:

Tipo	**Descrição**	**Tamanho em bytes**	**Faixa mínima**

char	caractere	1	-127 a 127
int	número inteiro	4	-32767 a 32767
float	número em ponto flutuante (real)	4	seis dígitos de precisão
double	número em ponto flutuante de dupla precisão	8	dez dígitos de precisão
void	sem valor	-	-

O tamanho em bytes da cada tipo de dado pode variar de acordo com o tipo do processador, da implementação do compilador, etc. Por exemplo, em Linux um inteiro possui 4 bytes, enquanto que no DOS um inteiro possui 2 bytes. O que o padrão ANSI estipula é apenas a faixa mínima de cada tipo, não o seu tamanho em bytes.

O tipo **void** é usado quando declaramos funções que não retornam nenhum valor, quando queremos declarar explicitamente que uma função não possui argumentos, ou para criar ponteiros genéricos.

- ***O que são Variáveis?***

Para você poder manipular dados dos mais diversos tipos, é necessário poder armazená-los na memória e poder referenciá-los quando for preciso. É para isso que existe as **variáveis**, que nada mais são do que um espaço reservado na memória, e que possuem um nome para facilitar a referência. As variáveis podem ser de qualquer tipo: int, char, float, double, void.

Como o próprio nome diz, as variáveis podem ter o seu conteúdo alterado durante a execução do programa, ou seja, o programador tem a liberdade de atribuir valores ao decorrer da execução.

- ***Declaração de Variáveis***

Todas as variáveis em C devem ser declaradas antes de serem usadas. A forma geral de declaração de uma variável é a seguinte:

```
tipo nome_da_variavel;
```

Exemplos:

Uma variável de cada vez:

```
int id;
```

```
float aux;
```

Ou várias do mesmo tipo em uma única linha:

```
int id, obj, n, t;
char c1, c2, c3;
```

Onde *tipo* pode ser qualquer tipo válido da linguagem C, e *nome_da_variavel* deve ser um nome dado pelo programador, sendo o primeiro caractere uma letra e os demais, letras, números ou "_".

NUM	CARAC
PESO	ID_OBJ
ALUNO_1	AUX

Lembrando que C diferencia MAIÚSCULAS de minúsculas.

Não são nomes válidos de variáveis:

1NUM	-IDADE
$AUX	ID@AL

- ***Inicializando Variáveis***

Inicializar significa atribuir um valor inicial a uma variável. Em C, podemos inicializar variáveis na declaração:

```
int n=12;
```

Neste exemplo, estamos declarando a variável inteira *n* e atribuindo o valor 12 a ela. O sinal de igual (=) em C é o operador de atribuição. Ele serve para colocar o valor (do lado direito) na variável (lado esquerdo).

Podemos também inicializar uma variável no corpo do programa, após sua declaração:

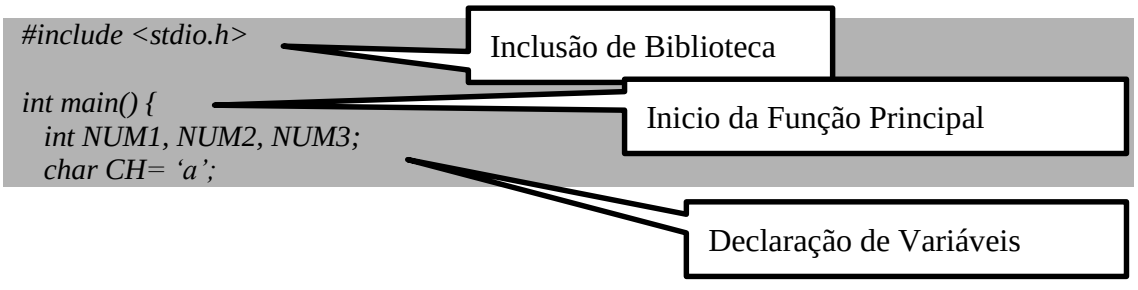
```
int n;
n=12;
```

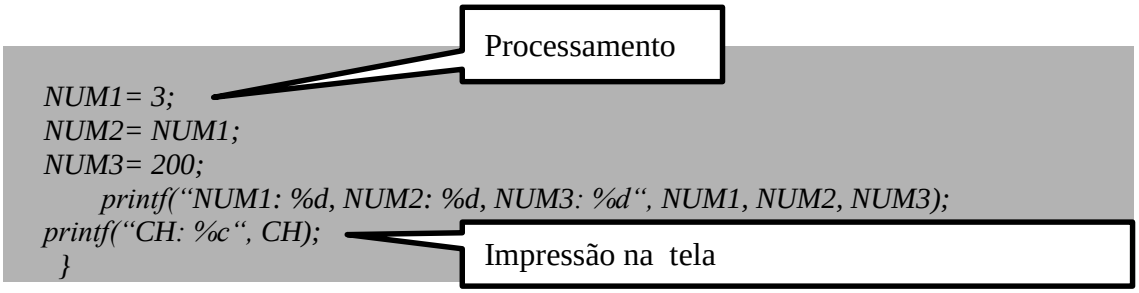
Pode-se atribuir também variáveis a variáveis:

```
num=i;
```

Neste caso, a variável *num* está recebendo o valor da variável *i.*

Para fixar melhor, aí vai um exemplo de um programa em C contendo variáveis:





- ***Constantes***

Constantes em C são valores fixos, que não mudam durante a execução. Elas podem ser de qualquer tipo básico: inteiras, ponto flutuante, caractere

Na maioria dos casos C escolhe o menor tipo de dado possível para tratar uma constante, com exceção das constantes em ponto flutuante, que são tratadas como double, por default.

Veja um exemplo de programa com constantes:

```
#include <stdio.h>

int main()
{
  char ch= 'a';
  int n= 10;
  float f= 10.4;

  printf("%c  %d  %f ", ch, n, f);

}
```

Saída - A Função printf

Para termos acesso à biblioteca que contém as funções, macros e variáveis que facilitam a entrada e saída de dados o programa deve conter a declaração.

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

A função printf permite que dados sejam escritos na tela do computador. O formato é ao mesmo tempo de uso simples e bastante flexível, permitindo que os dados possam ser apresentados de diversas maneiras.

A forma geral é:

```
printf (controle, arg1, arg2, ...);
```

onde os argumentos são impressos de acordo com a maneira indicada pelo controle. Um exemplo simples pode tornar a explicação mais clara.

O programa abaixo (ano.c) imprime o valor da variável ano.

```
#include <stdio.h> /* Biblioteca de entrada e saída */

int main( ){    /* Aqui começa o programa principal*/

        int ano = 2007;  /* Declarei ano como inteiro e já defini seu valor.  */

        printf("Estamos no ano %d.", ano); /* Imprime o valor do ano */


}
```

Na tela do computador será impresso:

```
Estamos no ano 2007.
```

O controle, que deve aparecer sempre entre " ", define como serão impressos os argumentos. Neste controle podem existir dois tipos de informações: caracteres comuns e códigos de formatação. Os caracteres comuns, como no exemplo **(Estamos no ano)** são escritos na tela sem nenhuma modificação. Os códigos de formatação aparecem precedidos por um % e são aplicados aos argumentos na ordem em que aparecem. Deve haver um código de formatação para cada argumento. O código %d indica que o valor armazenado em ano deve ser impresso na notação inteiro decimal.

- **Códigos de Conversão**

Os códigos de conversão são os seguintes:

Códigos de Conversão para leitura e entrada de dados.

Código	Comentário
%c	Caractere simples
%d	Inteiro decimal com sinal
%i	Inteiro decimal com sinal

%E	Real em notação científica com E
%e	Real em notação científica com e
%f	Real em ponto flutuante
%G	Use %E ou %f, o que for mais curto
%g	Use %e ou %f, o que for mais curto
%o	Inteiro em base octal
%s	Cadeia Caracteres
%u	Inteiro decimal sem sinal
%x	Inteiro em base hexadecimal (letras minúsculas)
%X	Inteiro em base hexadecimal (letras maiúsculas)
%p	Ponteiro
%%	Imprime o caractere %

Entre o % e o caractere do código podem ser inseridos os seguintes comandos:

Justificação da saída

Um sinal de menos para especificar que o argumento deve ser ajustado à esquerda no seu campo de impressão.

O programa a seguir (menos.c) ilustra os dois tipos de justificação.

```
#include <stdio.h>

int main( void)
{
   int ano = 2006;

  printf("Justificado para direita Ano = %8d\n", ano);
  printf("Justificado para esquerda Ano = %-8d\n",


                              ano);
   return 0;
 }
```

- ***Tamanho do Campo***

Um número inteiro especificando um tamanho de campo onde o argumento será impresso. No exemplo acima o número especifica que 8 espaços são reservados para imprimir o resultado.

- ***Especificador de Precisão***

O especificador de precisão consiste de um ponto que separa o número que define o tamanho do campo do próximo número. Este próximo número especifica o máximo número de caracteres de uma string a ser impresso, ou o número de dígitos a serem impressos a direita do ponto em um número do tipo ***float*** ou ***double.***

No exemplo abaixo (prec.c) o número calculado é impresso com 3 casas decimais em um campo de tamanho 9 espaços.

```
#include <stdio.h>

main()
{
   float r = 1.0/3.0;
   printf("O resultado e = %9.3f  \n", r);
}
```

Ao executar o exemplo verifique que \n não é impresso. A barra inclinada é chamada de sequência de escape, indicando que o próximo caractere não é para ser impresso, mas representa caracteres invisíveis ou caracteres que não estão representados no teclado, por exemplo. Alguns destes caracteres são:

Código	Comentário
\n	Passa para uma nova linha
\t	Tabulação
\b	Retorna um caractere
\f	Salta uma página
\0	Caractere nulo
\xhh	Caractere representado pelo código ASCII hh (hh em notação hexadecimal)
\nnn	Representação de um byte em base octal

Entrada - A Função scanf

A função scanf pode ser utilizada para entrada de dados a partir do teclado e seu formato é:

```
scanf(controle, arg1var1, arg2var2, ...);
```

Uma diferença fundamental existe entre esta função e a função printf:

- Os argumentos são os endereços das variáveis que irão receber os valores lidos e não, como em printf, as próprias variáveis.

- A indicação que estamos referenciando um endereço e não a variável se faz pelo operador &. Por exemplo, o comando:

```
scanf("%d %d", &a, &b);
```

Espera-se que dois valores inteiros sejam digitados no teclado. O primeiro armazenado na variável ***a*** e o segundo em ***b.***

Usualmente o campo de controle só contém especificações de conversão, que são utilizadas para interpretar os dados que serão lidos. No entanto outros caracteres podem aparecer. O campo de controle pode conter:

- Especificações de conversão, consistindo do caractere %, um caractere opcional de supressão da atribuição (caractere *), um número opcional para especificar o tamanho máximo do campo, e um caractere de conversão;

- Espaços, caracteres de tabulação ou *linefeeds* (tecla enter) que são ignorados;

- Caracteres comuns (não %) que devem casar com o próximo caractere diferente de branco da entrada.

Para que os valores digitados sejam separados por vírgulas, o comando deveria ser escrito da seguinte maneira:

```
scanf("%d, %f", &i, &x);
```

Observar que deve haver uma correspondência exata entre os caracteres não brancos do controle e os caracteres digitados. Neste caso a entrada deveria ser:

35, 46.3

Lendo e Imprimindo Caracteres

- ***Funções getchar e putchar***

Para ler e escrever caracteres do teclado, as funções de entrada e saída mais simples são ***getchar*** e ***putchar*** que estão na biblioteca ***stdio.h***, são os seguintes:

```
int getchar(void);
int putchar(int c);
```

Apesar da função ***getchar*** retornar um parâmetro inteiro é possível atribuir este valor a uma variável do tipo ***char*** porque o código do caractere está armazenado no byte de ordem mais baixa. O mesmo acontece com a função ***putchar*** que recebe um inteiro, mas somente o byte de ordem mais baixa é passado para a tela do computador.

O programa getcha.c abaixo mostra exemplos de uso destas funções.

```
#include  <stdio.h>

int main ()
{
   char c;
   int i;

   printf("Entre com um caractere entre 0 e 9.\n");
   c = getchar();

   printf("O caractere lido foi o:  ");
   putchar(c);

}
```

Funções getch e getche

Na definição original da função getchar a entrada é armazenada até que a tecla ENTER seja apertada. Com isto caracteres ficam em um buffer esperando para serem tratados. Em ambientes onde é necessário tratar o caractere imediatamente esta função pode não ser útil.

Muitos compiladores incluem funções para permitir entrada interativa de dados. As duas funções mais comuns são getch e getche e seus protótipos, que podem ser encontrados em conio.h, são os seguintes:

```
int getch(void);
int getche (void);
```

A função getch espera até que uma tecla seja apertada e retorna este valor imediatamente. O caractere lido não é ecoado na tela. A função getche opera da mesma maneira, mas ecoa o caractere lido.

Lendo e Imprimindo Strings

Um string em C é uma cadeia de caracteres. Para usar strings é preciso primeiro definir um espaço para armazená-los. Para isto é preciso declarar o tamanho e o tipo do vetor. Por exemplo:

```
char nome[40];     /* reserva um espaço para armazenar caracteres. */
```

Quando definir o tamanho do vetor observar que todo string em C termina com o caractere null (‘\0’), que é automaticamente inserido pelo compilador. Portanto o vetor nome pode armazenar no máximo 39 caracteres.

Após este passo a variável nome pode ser usado durante a execução do programa.

- ***Lendo e Imprimindo strings com scanf e printf***

O exemplo stscf.c mostrado abaixo mostra como ler e imprimir um string usando os comandos scanf e printf respectivamente.

```
#define DIM 40                                  /* Definições  Constantes*/
#include  <stdio.h>                          /* Biblioteca usada pelo programa */
void main (void)
{
   char nome[DIM];                        /* linha de caracteres lidos do teclado */
   printf(“Por favor, qual o seu nome.”);
   scanf(“%s”, &nome);                                /* Entrada de dados do vetor */
   printf(“Eu sou um computador PC, em que posso ajudá-lo %s? \n”, nome);
}
```

Caso o nome digitado tenha sido Antonio da Silva, o programa imprimiria somente o seguinte:

```
Eu sou um computador PC, em que posso ajudá-lo Antonio.
```

O código de conversão para strings é %s. Observar que o comando scanf não lê o nome todo, mas encerra a leitura quando encontra um caractere branco. Mas como ler para um vetor um nome inteiro, ou um string que contenha brancos? Para isto deve-se usar a função gets.

- ***Lendo e Imprimindo strings com gets e puts***

A função ***gets*** lê toda a string até que a tecla ENTER seja teclada, no vetor o código da tecla ENTER não é armazenado, sendo substituído pelo código null (‘\0’). Caso a função ***scanf*** do exemplo anterior fosse substituída pela ***gets*** o programa imprimiria:

```
Eu sou um computador PC, em que posso ajudá-lo Antonio da Silva.
```

O comando que substitui o scanf é gets(nome).

O protótipo da função gets é o seguinte:

```
#include<stdio.h>
char *gets (char *str);
```

A função puts tem o seguinte protótipo:

```
#include<stdio.h>
int puts (const char *str);
```

Ela imprime a string apontada por ***str***. O programa exemplo *stput.c,* mostrado abaixo, é semelhante ao exemplo anterior com as funções ***printf*** substituídas por ***puts***. Observe que a impressão sempre termina passando para a próxima linha.

```
#define DIM 40
#include <stdio.h>
void main ( )
{
   char nome[DIM];  /* linha de caracteres lidos do teclado */
   /* Entrada de dados do vetor */
   puts("Entre com o seu nome, por favor.");
   gets( nome);
   puts("Alo ");
   puts(nome);
   puts("Eu sou um computador PC, em que posso ajudá-lo?");
}
```

Operadores em C

O Operador de Atribuição

O operador de atribuição, (o símbolo de igual "=") coloca o valor de uma expressão (do lado direito) em uma variável (do lado esquerdo). Uma expressão neste caso pode ser um valor constante, uma variável ou uma expressão matemática mesmo.

É um operador binário, ou seja, trabalha com dois operandos. Exemplos:

Atribuição de uma constante a uma variável:

```
n= 10;
ch= 'a';
fp= 2.51;
```

Atribuição do valor de uma variável a outra variável:

```
n= num;
```

Atribuição do valor de uma expressão a uma variável:

```
n= (5+2)/4;
```

Atribuições múltiplas:

```
x = y = z = 20;
```

Em uma atribuição, primeiro é processado o lado direito. Depois de processado, então, o valor é atribuído a variável.

Como você viu no último exemplo acima, C também permite atribuições múltiplas (como x = y = z = 20;). Neste caso, todas as variáveis da atribuição (x, y e z) recebem o valor mais à direita (20).

Os Operadores Aritméticos

Estes são, de longe, os mais usados. Os operadores aritméticos em C trabalham praticamente da mesma forma que outras linguagens. São os operadores **+ (adição), - (subtração), * (multiplicação), / (divisão) e % (módulo ou resto da divisão inteira)**, todos esses binários (de dois operandos). Temos também o **- unário**, que muda o sinal de uma variável ou expressão para negativo. Veja a tabela a seguir:

Operador	Descrição	Exemplo
- unário	Inverte o sinal de uma expressão	-10, -n, -(5*3+8)
*	Multiplicação	3*5, num*i
/	Divisão	2/6, n/(2+5)
%	Módulo da divisão inteira (resto)	5%2, n%k
+	Adição	8+10, exp+num
-	Subtração	3-6, n-p

Uma expressão deste tipo:

```
5 + 2 * 3 - 8 / 4
```

É avaliada assim: primeiro a multiplicação (2*3), depois a divisão (8/4). Os resultados obtidos destas duas operações são utilizados para resolver as duas últimas operações:

```
5 + 6 – 2
```

Adição e subtração. Igualzinho à matemática aprendida no primário...

Tudo isso porque as operações de multiplicação e divisão têm maior precedência e são resolvidos primeiro em uma expressão. Para mudar a ordem de operação, devem-se usar parênteses. Deve-se tomar cuidado ao construir expressões, pois a falta de parênteses pode causar inconsistência (erro) no resultado.

O Operador **%** é equivalente ao **mod**, e é útil em várias situações. Ele dá como resultado o resto da divisão inteira de dois operandos. Assim, fica fácil, por exemplo, saber se um número é múltiplo de outro:

```
if ((num%3)==0)   /* se o resto da divisão entre num e 3 for igual a 0 ... */
        printf("Múltiplo de 3\n");
```

Este é apenas um de vários problemas que podem ser resolvidos com o uso do operador **%**.

Operadores Relacionais e Lógicos

Os operadores relacionais e lógicos são usados em testes e comparações, principalmente nos comandos de controle e nos laços.

Para entender melhor esses operadores, temos que entender o conceito de **verdadeiro** e **falso**. Em C, **verdadeiro é qualquer valor diferente de zero**, e **falso é zero**. As expressões que usam operadores relacionais e lógicos retornam 0 para falso e 1 para verdadeiro.

Os operadores relacionais são 6:

Operador	Ação
<	Menor que
<=	Menor que ou igual
>	Maior que
>=	Maior que ou igual
==	Igual
!=	Diferente

Veja um exemplo do uso de operadores relacionais:

```
#include <stdio.h>
          /* necessário para printf e scanf */

int main(){
         int n;       /* Declaração de uma variável inteira */
        printf("Digite um número: ");
```

```
        scanf("%d", &n);   /* Lê o número e armazena na  variável n */
        if (n < 0)  /* Se n for MENOR QUE 0... */
                printf("Número negativo\n");   /* ... escreve isto. */
                else  {  /* Senão... */
                printf("Número positivo\n");   /* ... escreve isto. */
                }
}
```

Tente fazer alguns testes com os outros operadores relacionais. Seja criativo! Você é o programador.

Os operadores lógicos são 3:

Operador	Ação	Formato da expressão
&&	and (**e** lógico)	p && q
||	or (**ou** lógico)	p || q
!	not (**não** lógico)	!p

Veja um exemplo, só do bloco *if*:

```
if ((n > 0) && (n < 100)) /* se n for > que 0 e n for < que 100 */
        printf("Número positivo menor que 100\n");  /* ...imprime isto */
```

Outro exemplo:

```
if ((n == 0) || (n == 1))    /* se n for IGUAL a 0 OU n for igual a 1 ... */
        printf("zero ou um\n");       /* ... imprime isto. */
```

Os parênteses também podem ser usados para mudar a ordem de avaliação das expressões, como no caso das expressões aritméticas.

Operadores de Incremento e Decremento

A linguagem C possui dois operadores que geralmente não são encontrados em outras linguagens, mas que facilita bastante a codificação: os operadores de incremento(++) e decremento(--).

O operador ++ soma 1 ao seu operando, similar a operação de:

```
variável= variável+1;
```

mas muito mais resumido.

O operador - - subtrai 1 de seu operando, também similar a:

```
variável= variável-1;
```

Estes operadores são unários, e podem ser usados antes da variável:

```
++n;
```

ou depois da variável:

```
n++;
```

A diferença é que, se o operador precede o operando (++n), o incremento ou decremento é realizado antes do valor da variável ser usado. E se o operador vem depois do operando (n++), o valor da variável poderá ser usado antes de acontecer à operação de incremento ou decremento. Veja estes dois exemplos:

Exemplo 1:

```
n= 5;
p= ++n;
printf("%d ",p);                /* imprime na tela: 6 */
```

Exemplo 2:

```
n= 5;
p= n++;
printf("%d ",p);                /* imprime na tela: 5 */
```

No exemplo 1, a variável n é incrementada de 1 ANTES de seu valor ser atribuído a p. No exemplo 2, o valor de n é atribuído a p antes de acontecer à operação de incremento. Essa é a diferença de se colocar esses operadores antes ou depois da variável.

Operadores Aritméticos de Atribuição

Algumas operações de atribuição podem ser feitas de forma resumida. Por exemplo, a atribuição:

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

pode ser escrita:

```
x+=10;
```

A forma geral desse tipo de atribuição é:

```
variável [operador]= [expressão];
```

que é igual a:

```
variável= variável [operador] [expressão]
```

Veja essa tabela com exemplos:

Forma longa	Forma resumida
x= x+10	x+=10
x= x-10	x-=10
x= x*10	x*=10
x= x/10	x/=10
x= x%10	x%=10

Se familiarize com essa notação, pois é um estilo largamente adotado pelos profissionais que trabalham com C.

Estruturas de Decisão

Os comandos para tomadas de decisão em C são: if, if ... else, switch e o operador ternário (?).

O comando if

O comando if, presente na maioria das linguagens de programação, é usado para a construção de estruturas de decisão simples. A forma geral do comando é assim:

```
if (expressão condicional)
{
  comandos;
}
```

OU

```
if (expressão condicional)
  comando único;
```

O comando if avalia a expressão condicional entre parênteses. Se esta expressão for verdadeira, o(s) comando(s) do bloco de código será(ão) executado(s). Se for falsa, o bloco de código do if não será executado, isto é, o programa "passa batido".

Para comandos if com apenas UM comando, não é necessário delimitar o bloco de código com chaves ("{}"). O compilador sabe que quando não existem as chaves naquele if, somente UM comando fará parte do bloco de código.

Agora, vamos ver alguns exemplos:

```
if (a > b)   /* se o a for maior do que o b ... */
    printf("a é maior que b");   /* ... imprime mensagem */
```

```
if ((num % 2) == 0)    /* se o resto da divisão entre num e 2 for 0 ... */
{
   printf("Número par"); /*  ... faça   isso */
   printf("\n\n");
}
```

É interessante sempre manter o código identado, com tabulações, para ficar claro de qual bloco de código é determinado comando.

O comando if ... else

Esta construção possui duas partes: um comando ***if***, cujo bloco de código será executado se a expressão condicional for *verdadeira*; e um comando ***else***, que será executado caso a mesma expressão seja *falsa*.

Portanto, agora temos uma "bifurcação", e o caminho que o programa trilhará vai depender do resultado da expressão condicional (verdadeiro ou falso).

Forma geral:

```
if (expressão condicional)
{
    comandos;
}
else  {
    comandos;  }
```

Se os blocos de código tiverem mais que um comando, deve-se delimitá-los com abre e fecha chaves. Vejam alguns exemplos:

```
if ((num % 2) == 0)      /* Se o resto da divisão entre num e 2 for 0... */
    printf("Número par.");        /* ...imprime mensagem "número par". */
    else {  /* ... Senão... */
    printf("Número ímpar."); /* ... imprime mensagem "número ímpar". */
    }
```

Neste exemplo, se num % 2 for igual à zero (se esta expressão for verdadeira), executa o primeiro printf(). Caso contrário (se for falsa), executa o segundo printf().

Dentro de um bloco de código, pode aparecer qualquer comando, até mesmo if's e else.

Você pode até criar construções para decisões múltiplas, criando uma espécie de escada de if's e else.

O Comando switch

O comando switch é usado como alternativa à escada de if's e else's, e torna o código mais elegante e legível. Forma geral:

```
switch (expressão constante)
{
        case constante1:    comandos;
        break;
        case constante2:    comandos;
        break;  `
        case constante3:    comandos;
        break;
        case constante4:    comandos;
        break;
        default:            comandos;
}
```

Este comando possui algumas restrições. Como a expressão é constante, ele só aceita valores ***inteiros*** e ***caracteres.*** Como expressão condicional, só são aceitas igualdades (se a constante1 for igual à expressão constante, executa comandos; e assim por diante).

Para cada case existe um conjunto de comandos que só serão executados caso a condição seja verdadeira. Se a expressão constante não casar com nenhum case, então os comandos do bloco ***default*** são executados. O bloco de código dos case NÃO PRECISAM ser delimitados com chaves; basta colocá-los, um após o outro, após o rótulo case:

Vamos ver um exemplo para poder "clarear":

```
#include <stdio.h>
int main()
{
    float n1, n2;
    char op;

    printf("*** Calculadora ***\n");
    printf("Digite um número: ");
    scanf("%f", &n1);

    printf("Digite o operador (+ - * /): ");
    scanf("%c", &op);

    printf("Digite outro número: ");
    scanf("%f", &n2);

    switch(op)                          /*expressão constante: variável op (caractere) */
    {
       case '+':
                    printf("Adição: %.2f\n", n1+n2);
                    break;
       case '-':
                    printf("Subtração: %.2f\n", n1-n2);
                    break;
       case '*':
                    printf("Multiplicação: %.2f\n", n1*n2);
                    break;
       case '/':
                    if (n2 == 0)
                    {
                    printf("Erro: impossível dividir por zero!\n");
                    return 1;
                    }else {
                    printf("Divisão: %.2f\n", n1/n2);
                    }
                    break;
          default:
                    printf("Operador inválido: %c\n", op);
                    break;
    }

}
```

Um exemplo do ***switch*** na prática. Em cada ***case***, temos o(s) comando(s) e, em seguida, um comando ***break***. O comando ***break***, quando encontrado, faz a execução sair imediatamente do ***switch***. Se usado em loop's, faz a execução do programa parar o loop e continuar a execução do restante do código após o loop.

Imagina se, no lugar do ***switch***, tivéssemos que montar uma escada de ***if's*** e ***else's***... Ficaria mais ou menos assim:

```
if (op=='+')
```

```
        printf("Adição: %.2f\n", n1+n2);
        else  if (op=='-')
                printf("Subtração: %.2f\n", n1-n2);
                 else  if (op=='*')
                        printf("Multiplicação: %.2f\n", n1*n2);
                         else  if (op=='/'){
                                if (n1==0)
                                        printf("Erro: impossível dividir por zero!\n");
                                         else printf("Divisão: %.2f\n", n1/n2);
                                } else  printf("Operador inválido: %c\n", op);
```

Usando o ***switch***, neste exemplo, temos 25 linhas de código, enquanto que usando a construção de ***if's*** e ***else's*** temos 11 linhas.

O ***switch*** é bastante usado em menus, quando o usuário tem que digitar o número ou a letra de alguma opção.

Estruturas de Repetição

Suponhamos que você queira fazer um programa que lê 100 números do teclado e para cada número verifique se é múltiplo de 3, imprimindo a resposta para cada um.

Como você escreveria este programa? Seria assim?

```
#include <stdio.h>
main(){
 int num;

 printf("Digite um número: ");
 scanf("%d", &num);

 if ((num %3) == 0)
        printf("é múltiplo de 3\n");
        else  printf("não é múltiplo de 3\n");

 printf("Digite um número: ");
 scanf("%d", &num);

 if ((num %3) == 0)
        printf("é múltiplo de 3\n");
        else    printf("não é múltiplo de 3\n");

 printf("Digite um número: ");
 scanf("%d", &num);

 if ((num %3) == 0)
        printf("é múltiplo de 3\n");
        else printf("não é múltiplo de 3\n");
```

```
printf("Digite um número: ");
scanf("%d", &num);

if ((num %3) == 0)
        printf("é múltiplo de 3\n");
        else  printf("não é múltiplo de 3\n");

printf("Digite um número: ");
scanf("%d", &num);

if ((num %3) == 0)
        printf("é múltiplo de 3\n");
        else printf("não é múltiplo de 3\n");
.
.
.
etc .....................
}
```

Como você pode ver, não é uma técnica nada viável, pois temos que repetir a mesma sequência de comandos várias vezes. Mas não se desespere o C, assim como outras linguagens, tem vários comandos que nos ajudam em tarefas repetitivas, como esta. Estes comandos são conhecidos como *estruturas de repetição* ou *laços de repetição*. Veremos também algumas técnicas básicas que nos ajudarão a criar programas iterativos.

O laço for

O laço ***for*** é uma estrutura de repetição controlada por variável, onde devemos saber de antemão a quantidade de iterações. O laço ***for*** tem a seguinte forma geral:

```
for ( inicialização ;  teste condicional ;  incremento ou decremento )
{
                comandos;
}
```

OU

```
for ( inicialização ;  teste condicional ;  incremento ou decremento )
                comando único;
```

Veja este exemplo:

```
#include <stdio.h>
main()
{
  int i, num;

  for (i=0; i<100; i++)
  {
    printf("Digite um número: ");
    scanf("%d", &num);
```

```
    if ((num %3) == 0)
      printf("é múltiplo de 3\n");
    else
      printf("não é múltiplo de 3\n");
  }
 }
```

Veja só o nosso exemplo revisitado, com o uso de ***for***.

Como você pode ver, o laço ***for*** contém 3 expressões entre parênteses, que controlarão o modo que o nosso laço trabalhará:

Na expressão inicialização, atribuímos valor inicial à variável de controle. No exemplo acima, *inicializamos i com 0.*
Em *teste condicional*, temos uma expressão relacional/lógica que definirá o critério de parada do nosso laço. No nosso //*Detector de múltiplos de 3*//, o laço ficará repetindo o bloco de comandos ***enquanto a variável i for menor que 100***. Quando ***i*** tiver o valor ***100***, o laço não executará mais os comandos do bloco de comandos, e o programa continuará sua execução (no nosso caso, executará ***return0;*** e depois terminará).

A expressão ***incremento*** ou ***decremento*** deve dizer para o laço ***for*** como a variável vai "*andar*", se é de 1 em 1, de 2 em 2, de 10 em 10, etc. No nosso programa acima, a expressão ***i++*** faz a variável ***i*** ser incrementada de 1 em 1.

Trocando em miúdos, no laço ***for*** do nosso programa, a variável ***i*** será inicializada com 1, andará de 1 em 1, e quando chegar a 100 sairá do laço.

- ***Flexibilidade do laço for***

As 3 expressões do laço podem conter várias instruções separadas por vírgula. Vamos ver um exemplo:

```
#include <stdio.h>
main()
{
  int i, j, num;

          for (i=0, j=0; i+j <100; i++, j++)
    printf("%d\n", i+j);
    return 0;
}
```

Veja que na inicialização, temos duas variáveis sendo atribuídas com 0: ***i e j***, separadas por vírgula. Na parte de incremento, ***i e j*** são incrementadas com 1, também separadas por vírgula. Isto torna o laço ***for*** bastante flexível, onde o limite é a sua criatividade. Mas cuidado para não construir laços ***for*** monstros, com expressões enormes e de difícil compreensão...

Além disso, podemos omitir qualquer uma das expressões do laço ***for***. Por exemplo:

```
for ( ; i<100; i++)
{
  comandos;
}
```

Nesse caso, não haverá a inicialização.

Outro exemplo:

```
for (i=0; i<100; )
{
  comandos;
}
```

Aqui, não há o incremento. Ele pode ser feito dentro do bloco de código do laço.

Em alguns casos, podemos usar o laço ***for*** como um laço infinito. Veja como:

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

Este é um laço ***for*** infinito. Sua execução nunca terminará, a menos que haja um ***return*** ou um ***break*** dentro dele.

O laço while

O laço ***while*** é uma estrutura de repetição que só repetirá o bloco de código se a condição, entre parênteses, for verdadeira. Esta condição pode ser qualquer expressão, onde *falso é 0* e *verdadeiro é diferente de 0*. Se a condição for falsa logo no início do ***while***, o laço não será executado.

Forma geral:

```
while (condição)
{
      comandos;
}
```

OU

```
while (condição)
```

```
        comando único;
```

Quando o programa chega ao ***while***, ele verifica a condição entre parênteses. Se for verdadeira, ele executa o bloco de códigos. Quando termina de executar os comandos do bloco de código, ele volta para o ***while*** novamente, testa a condição de novo e, se for verdadeira, executa o bloco de código novamente. Faz isso até a condição entre parênteses ser falsa, momento em que o programa continua a execução normalmente, depois do ***while***.

Vamos ver alguns exemplos:

```
#include <stdio.h>
#include <ctype.h>
main()
{
  int cont=0, i= 0;
  char ch;
  while (i < 10)
  {
    printf("Digite um caractere: ");
    scanf("%c", &ch);
    getchar();
    if (isdigit(ch))
   {
      cont++;
      i++;
         }
  }
 printf("Você digitou %d caracteres numéricos\n", cont);
 return 0;
}
```

Aqui, o programa utiliza a função ***isdigit()***, presente no arquivo de cabeçalho *ctype.h*. O *ctype.h* contém funções para teste e conversão de caracteres.

Neste programa, o ***while*** executará 10 vezes, só terminando quando ***i*** chegar ao valor 10. Aqui vemos uma técnica que usamos muito em várias aplicações: um contador. A variável ***cont*** é inicializada com zero; quando, dentro do ***while***, o cara digitar um caractere numérico, esta variável é incrementada de 1. Ao final da execução, ela terá o número exato de caracteres numéricos que foram digitados.

Outro exemplo:

```
#include <stdio.h>
int main()
{
  int i=0, num, res= 0;
  while (i < 5)
  {
    printf("Digite um número: ");
    scanf("%d", &num);
    res= res+num;
    i++;
```

```
  }
  printf("Somatória: %d\n", res);
  return 0;
}
```

Neste, outra técnica bastante utilizada: a somatória de um conjunto de números.

Funciona assim: ao entrar no ***while***, o cara digita um número, que será armazenado na variável ***num***. Esse número é somado com o valor da variável ***res*** (que na primeira iteração é 0) e armazenado em ***res***. Na próxima iteração, o valor da variável res (a soma da iteração passada) é somada com num e armazenada em res novamente, assim sucessivamente até acabar o laço. Ao final do laço, o que temos em ***res*** é a somatória de todos os números digitados.

Mais uma técnica: imagine que você queira somar os números digitados até que seja digitado um número maior que 100. Você pode criar uma variável que muda de valor quando essa condição for satisfeita. É o que chamamos de ***flag***. Exemplo:

```
int num;
int flag= 0;
while (flag == 0)
{
   printf("Digite um número: ");
   scanf("%d", &num);
   getchar();
        if (num > 100)
     flag= 1;                    /* flag muda de valor (indica que loop deve parar) */
}
```

Lembrando que estas técnicas podem ser exploradas em qualquer tipo de laço, desde que sejam adaptadas para cada situação.

Assim como ***for***, podemos fazer um laço infinito com o ***while*** também. Basta fazer o seguinte:

```
while(1)        /* 1 é verdadeiro em C */
{
  comandos;
}
```

É claro que em *comandos* deve haver alguma forma de sair do ***while***, como um ***break***, um ***return***, um ***exit()***, etc.

OBS: Evite usar comandos como ***break*** ou ***exit()*** dentro de laços, pois é uma prática pouco elegante. Só use quando não houver outro jeito.

O laço do-while

Diferente do ***while***, que testa a condição antes de executar o seu bloco de código, o ***do-while*** primeiro executa o seu bloco de código para depois testar a condição. Ele é útil quando temos que executar no mínimo UMA VEZ o bloco de código do laço.

Forma geral:

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

No caso de haver um comando único, não é necessário delimitar com chaves, mas é recomendável, para não haver confusão e para deixar o código mais legível.

Vamos para os exemplos:

```
#include <stdio.h>
main()
{
    int num;
    do
    {
        printf("\n\nDigite um numero: ");
                scanf("%d", &num);
                getchar();
    } while ( num < 50 );
    return 0;
}
```

Neste programa, que não faz nada de útil, o bloco de código deve ser executado pelo menos uma vez, já que dependemos do valor da variável num para processar a condição, no final do laço. Aqui, o programa só sai do laço quando for digitado algum número maior ou igual a 50.

Agora um exemplo mais prático:

```
#include <stdio.h>
main()
{
  int opt;

  do
  {
    do
    {
      printf("**** Menu ****\n");
      printf("1) Cadastrar\n");
      printf("2) Buscar\n");
      printf("3) Exibir Relatório\n");
      printf("4) Sair\n");
      printf("Digite sua opção: ");
      scanf("%d", &opt);
      getchar();
          } while ((opt < 1) || (opt > 4));/* o cara fica preso até digitar um número válido */
            switch(opt)
      {
```

```
        case 1:         cadastra();
          break;
        case 2:         busca();
          break;
        case 3:         relat();
          break;
      }
   } while (opt != 4);     /* laço principal, só sai quando opt for igual a 4 */
   return 0;
}
```

Como você pode ver, temos dois ***do-while***. O mais externo e principal fica vigiando se o cara digitou 4 (se quer sair). Caso contrário, volta e executa tudo novamente.

O laço mais interno é uma técnica usada quando o usuário tem um menu, e tem que digitar uma opção para definir para que lado o programa vai. No nosso caso, se o espertinho do usuário digitar algum número maior que 4 ou menor que 1 (inválidos), o laço volta e joga tudo de novo na tela, esperando o esperto do usuário escolher a opção certa.
As funções ***cadastra(), busca() e relat()*** são fictícias e só serviram para ilustrar a situação.

Laços aninhados

Laços aninhados não são laços dentro de ninhos. São laços dentro de laços.

Exemplo de laços ***for*** aninhados:

```
for ( i = 1 ;i <= 10; i++ )
{
  for ( j = 1 ; j <= 10 ;  j++ )
  {
        printf("%d X %d = %d\n", i, j, i*j);
  }
  printf("\n\n");
  ch= getchar();
}
```

Vejam que poder: com apenas 9 linhas imprimimos as tabuadas de 1 a 10!!

Exemplo de laços ***while*** aninhados:

```
while ( i <= 10 )
{
  while ( j <= 10 )
  {
        printf("%d x %d = %d\n", i, j, i*j);
    j++;
  }
  printf("\n\n");
  ch= getchar();
```

```
  i++;
}
```

O mesmo exemplo das tabuadas usando ***while***.

A lógica dos laços aninhados é a seguinte:

- O laço mais externo entra em execução;
- São executados os comandos do bloco até que o programa encontra o laço mais interno;
- O laço mais interno também entra em execução;
- Agora o laço mais interno fará todas as iterações possíveis, até acabar;
- Acabando a execução do laço mais interno, o laço mais externo continua executando seu bloco de código;
- Quando o último código do bloco de código é executado, o laço mais externo volta, testa e entra em execução novamente(se a condição for satisfeita);
- Aí encontra o laço mais interno novamente, que entra em execução até terminar;
- E assim vai, até a condição de parada do laço mais externo ser satisfeita (the end).

Podemos ter vários laços aninhados, não somente 2. Podemos também ter laços ***for*** dentro de ***while*** ou ***do-while***, e etc... Só tem uma coisa: quanto mais laços aninhados, mais lento será seu programa.

Extraído do Material: M06 – Introdução a Programação, 2007, SIMÕES, S.; FONSECA, S. C. C.; MICRONI, A. D. L.; MICRONI, J. S. C; RODOVALHO, R. S.