Curso Linguagem C Ice2642 por Marcio Esper.

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

Linguagem feita para programadores.

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

Modulo 1.

Importante:
A linguagem C é case sensitive, isto quer significa que letras maiúsculas e minúsculas são tratadas como caracteres diferentes. A != a (A é diferente de a).

Programação Estruturada.

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

Declaração de variáveis.

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

Funções - Blocos de código.

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

Laços.

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

Testes de condições.

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

Programa de praxe. Exemplo de programa em C.

```
// Este é o primeiro programa

#include <stdio.h>

main()
   {
       printf("Alo mundo\n");
   }
```

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

1 – Comentário, toda linha iniciada por // se torna um comentario. Os comentarios são importantes dentro de um programa complexo pois nos ajuda a sentrar a atenção no foco e recordarmos determinados trechos que no decorrer do programa podem se tornar confusos.

Outra forma de se usar comentarios quando estes tem varias linhas ou um padrão é usar **/*** e ***/**

2 – include <nome da biblioteca> Esta é uma forma de emendarmos partes já programadas em nosso programa para aliviar o trabalho. Uma biblioteca é um conjunto de comandos, funções ou definições que podem ser usados repetidamente, então ao invez de se escrever tudo a cada vez que se faz um novo programa, podemos criar bibliotecas que serão usadas em diversos programas. Existem bibliotecas padrão e podemos criar quantas bibliotecas quisermos para infitos fins.

No caso a usada neste primeiro programa foi a stdio.h

Note que enquanto programamos em **C** toda linha termina com um ponto e virgula ";" só que o #include<> não segue esta regra.

A biblioteca **stdio .h** Define trez tipos de variáveis, e diversos macros e funções para entrada e saida de dados.

No standard library encontramos as seguintes explanações para o stdio. Especificados a seguir.

Library Variables

Following are the variable types defined in the header stdio.h:

S.N.	Variable & Description
1	**size_t** This is the unsigned integral type and is the result of the **sizeof** keyword.
2	**FILE** This is an object type suitable for storing information for a file stream.
3	**fpos_t** This is an object type suitable for storing any position in a file.

Library Macros

Following are the macros defined in the header stdio.h:

S.N.	Macro & Description
1	**NULL** This macro is the value of a null pointer constant.
2	**_IOFBF, _IOLBF** and **_IONBF** These are the macros which expand to integral constant expressions with distinct values and suitable for the use as third argument to the **setvbuf** function.
3	**BUFSIZ** This macro is an integer which represents the size of the buffer used by the **setbuf** function.
4	**EOFM** This macro is a negative integer which indicates an end-of-file has been reached.
5	**FOPEN_MAX** This macro is an integer which represents the maximum number of files that the system can guarantee that can be opened simultaneously.
6	**FILENAME_MAX** This macro is an integer which represents the longest length of a char array suitable for holding the longest possible filename. If the implementation imposes no limit, then this value should be the recommended maximum value.
7	**L_tmpnam** This macro is an integer which represents the longest length of a char array suitable for holding the longest possible temporary filename created by the tmpnam function.
8	**SEEK_CUR, SEEK_END,** and **SEEK_SET** These macros are used in the **fseek** function to locate different positions in a file.
9	**TMP_MAX** This macro is the maximum number of unique filenames that the function tmpnam can generate.
10	**stderr, stdin,** and **stdout** These macros are pointers to FILE types which correspond to the standard error, standard input, and standard output streams.

Library Functions

Following are the functions defined in the header stdio.h:

Follow the same sequence of functions for better understanding and to make use of **Try it**option

because file created in the first function will be used in subsequent functions.

S.N.	Function & Description
1	int fclose(FILE *stream) Closes the stream. All buffers are flushed.
2	void clearerr(FILE *stream) Clears the end-of-file and error indicators for the given stream.
3	int feof(FILE *stream) Tests the end-of-file indicator for the given stream.
4	int ferror(FILE *stream) Tests the error indicator for the given stream.
5	int fflush(FILE *stream) Flushes the output buffer of a stream.
6	int fgetpos(FILE *stream, fpos_t *pos) Gets the current file position of the stream and writes it to pos.
7	FILE *fopen(const char *filename, const char *mode) Opens the filename pointed to by filename using the given mode.
8	size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream) Reads data from the given stream into the array pointed to by ptr.
9	FILE *freopen(const char *filename, const char *mode, FILE *stream) Associates a new filename with the given open stream and same time closing the old file in stream.
10	int fseek(FILE *stream, long int offset, int whence) Sets the file position of the stream to the given offset. The argument *offset* signifies the number of bytes to seek from the given *whence* position.
11	int fsetpos(FILE *stream, const fpos_t *pos) Sets the file position of the given stream to the given position. The argument *pos* is a position given by the function fgetpos.
12	long int ftell(FILE *stream) Returns the current file position of the given stream.
13	size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream) Writes data from the array pointed to by ptr to the given stream.
14	int remove(const char *filename) Deletes the given filename so that it is no longer accessible.
15	int rename(const char *old_filename, const char *new_filename) Causes the filename referred to by old_filename to be changed to new_filename.
16	void rewind(FILE *stream) Sets the file position to the beginning of the file of the given stream.
17	void setbuf(FILE *stream, char *buffer) Defines how a stream should be buffered.
18	int setvbuf(FILE *stream, char *buffer, int mode, size_t size) Another function to define how a stream should be buffered.

19	FILE *tmpfile(void) Creates a temporary file in binary update mode (wb+).
20	char *tmpnam(char *str) Generates and returns a valid temporary filename which does not exist.
21	int fprintf(FILE *stream, const char *format, ...) Sends formatted output to a stream.
22	int printf(const char *format, ...) Sends formatted output to stdout.
23	int sprintf(char *str, const char *format, ...) Sends formatted output to a string.
24	int vfprintf(FILE *stream, const char *format, va_list arg) Sends formatted output to a stream using an argument list.
25	int vprintf(const char *format, va_list arg) Sends formatted output to stdout using an argument list.
26	int vsprintf(char *str, const char *format, va_list arg) Sends formatted output to a string using an argument list.
27	int fscanf(FILE *stream, const char *format, ...) Reads formatted input from a stream.
28	int scanf(const char *format, ...) Reads formatted input from stdin.
29	int sscanf(const char *str, const char *format, ...) Reads formatted input from a string.
30	int fgetc(FILE *stream) Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.
31	char *fgets(char *str, int n, FILE *stream) Reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.
32	int fputc(int char, FILE *stream) Writes a character (an unsigned char) specified by the argument char to the specified stream and advances the position indicator for the stream.
33	int fputs(const char *str, FILE *stream) Writes a string to the specified stream up to but not including the null character.
34	int getc(FILE *stream) Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.
35	int getchar(void) Gets a character (an unsigned char) from stdin.
36	char *gets(char *str) Reads a line from stdin and stores it into the string pointed to by str. It stops when either the newline character is read or when the end-of-file is reached, whichever comes first.

37	int putc(int char, FILE *stream) Writes a character (an unsigned char) specified by the argument char to the specified stream and advances the position indicator for the stream.
38	int putchar(int char) Writes a character (an unsigned char) specified by the argument char to stdout.
39	int puts(const char *str) Writes a string to stdout up to but not including the null character. A newline character is appended to the output.
40	int ungetc(int char, FILE *stream) Pushes the character char (an unsigned char) onto the specified stream so that the this is the next character read.
41	void perror(const char *str) Prints a descriptive error message to stderr. First the string str is printed followed by a colon then a space.

Main()

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

Note que primeira linha é figurativo, o main pode estar em qualquer parte do programa. No lite-C ele deve ser a ultima função do programa, ou será preciso o uso de protopiticos de função antes dele para que o compilador na apresente erro na depuração.

Como falado anteriormente, o C é executado com blocos de codigo, logo a seguir do mais uma chave abre o bloco **{** e uma chave termina ele **}**. todos os comandos e variáveis dentro deste bloco pertencem ao ploco, são locais e apenas vistas dentro deste bloco. O bloco main() vai ser tudo que esta entre as chaves, no caso do programa exemplo o comando printf. Um bloco completo que tem um proposito é chamado de função.

printf(“Alo mundo\n”);

O comando printif pertence a biblioteca stdio.h como apresentado assima, você vai notar então que ele nada mais é do que a chamada de uma função printif que tem como objetivo a saida de caracteres na tela.

No C podemos criar programas e comandos para criarmos estes programas de forma facil e organizada, consumindo com isso o minimo de uso do sistema que uma linguagem pode alcanção com exeção do proprio assembler. Por isso C é considerada uma linguagem de medio nível.

Em outras palavras, nos podemos programar a propria linguagem para conseguirmos alcansar

nossos objetivos, adicionando comandos que não vem por padrão na linguagem.

Printif (parametros e formato)

```
int printf(const char *format, ...)
```

format-- This is the string that contains the text to be written to stdout.It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags prototype is**%[flags][width][.precision][length]specifier**, which is explained below:

specifier	Output
c	Character.
d or i	Signed decimal integer
e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character
f	Decimal floating point
g	Use the shorter of %e or %f.
G	Use the shorter of %E or %f
o	Signed octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
p	Pointer address
n	Nothing printed.
%	Character.

flags	Description
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign..
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

width	Description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	Description
.number	For integer specifiers (d, i, o, u, x, X): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E and f specifiers: this is the number of digits to be printed after de decimal point. For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all

	characters are printed until the ending null character is encountered. For c type: it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

length	Description
h	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).
l	The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g and G).

•**additional arguments** -- Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter, if any. There should be the same number of these arguments as the number of %-tags that expect a value.Demonstração de uso da função printif

#include <stdio.h> //chamada da biblioteca.

```
int main ()
{
   int ch;

   for( ch = 75 ; ch <= 100; ch++ ) { //o comando for veremos mais para frente.
      printf("ASCII value = %d, Character = %c\n", ch , ch );
   }

   return(0);
}
```

Gera a saída a seguir.

```
ASCII value = 75, Character = K
ASCII value = 76, Character = L
ASCII value = 77, Character = M
ASCII value = 78, Character = N
ASCII value = 79, Character = O
ASCII value = 80, Character = P
ASCII value = 81, Character = Q
ASCII value = 82, Character = R
ASCII value = 83, Character = S
ASCII value = 84, Character = T
ASCII value = 85, Character = U
ASCII value = 86, Character = V
ASCII value = 87, Character = W
ASCII value = 88, Character = X
ASCII value = 89, Character = Y
ASCII value = 90, Character = Z
ASCII value = 91, Character = [
ASCII value = 92, Character = \
ASCII value = 93, Character = ]
ASCII value = 94, Character = ^
ASCII value = 95, Character = _
ASCII value = 96, Character = `
ASCII value = 97, Character = a
ASCII value = 98, Character = b
ASCII value = 99, Character = c
ASCII value = 100, Character = d
```

Importante lembrar.

Blocos de código

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

Ponto e vírgula.

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

Chaves.

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

Palavras reservadas.

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

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

Variáveis, constantes, operadores e expressões.

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

Identificadores.

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

Tipos de dados.

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

Tipo	Extensão em bits	Escala
char	8	128 a 127
int	16	31768 a 32767
float	32	32 3.4E-38 a 3.4E+38
doubl	64	64 1.7E-308 a 1.7E308
void	0	sem valor

Modificadores de tipo.

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

Modificadores de tipo.

Tipo	Extensão em bits
cha	8
int	16

short int	16
long int	32
float	32
short float	32
long float	64
double	64
short double	64
long double	80

Declarando variáveis.

Uma declaração de variável deve seguir a seguinte regra:

tipo lista_variáveis;

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

Exemplos de declaração de variável

```
int i, j, l;

short int si;

unsigned int ui;

long inteiro_grande;

double balanco, lucro, prejuizo;
```

Onde declarar?

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

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

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

Parâmetro (argumento) - O último lugar onde as variáveis podem ser declaradas é na declaração dos parâmetros formais de uma função, embora as variáveis aqui declaradas sejam utilizadas para

receber os argumentos quando a função é chamada, eles podem ser utilizados como outra variávei qualquer.

Programa exemplo:

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

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

display()
{
 int cont; /* Variável local, esta variável cont é diferente daquela declarada em main() */
 for(cont = 0; cont < 10; cont ++)
printf("-");
printf("A soma atual é %d", soma);
```

}

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

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

```
int soma = 0;
```

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

Constantes.

Constantes são valores fixos que o programa não pode alterar.

Tipo de dado	Exemplo
char	‘a’ ‘n’ ‘9’
int	1 123 2100 -234
long int	35000 -34
short int	10 -12 90
unsigned int	10000 987 40000
float	123.23 4.34E-3

double	123.23 12312333 -0.9876

Constantes hexadecimais e octais - Podem ser declaradas constantes em hexadecimal ou octal conforme o exemplo a seguir:

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

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

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

Código	Significado
\b	retrocesso
\f	avanço de página
\n	mudança de linha
\r	retorno de carro
\t	ab horizontal
\”	aspas duplas
\'	aspas simples
\0	ASCII 0
\\	barra invertida
\v	tab vertical
\a	Beep
\x	constante hexadecimal

Operadores.

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

Operadores aritméticos

Operador	Ação
-	subtração, também menos unário
+	adição
*	multiplicação
/	divisão
%	resto da divisão
--	decremento
++	incremento

Exemplo:

```
main()
{
        int x = 10, y = 3;
        printf(“%d\n”, x / y); // exibirá 3
        printf(“%d\n”, x % y); // exibirá 1, o resto da divisão de inteiros
        x = 1; y = 2;
        printf(“%d %d\n”, x / y, x % y); // exibirá 0 e 1
        x ++;
        printf(“%d\n”, x); // exibirá 2
        printf(“%d %d\n”, x++, ++y);
/* exibirá 2 e 3, neste caso, x só é incrementado depois que o comando é executado enquando y é incrementado antes */
}
```

++x e x++ são diferentes. No primeiro caso 1 sera adicionado a variavel antes do valor da variavel ser apontado, no segundo caso, depois. O mesmo se passa com o operador --.

Operadores de relação

Operador	Ação
>	maior que
>=	maior ou igual a
<	menor que
<=	menor ou igual a
==	igual
!=	não igual

Operadores lógicos

Operador	*Ação*
&&	AND
||	OR
!	NOT

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

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

Por exemplo.

```
int a, b, c; a = b = c = 1;

// atribui 1 às 3 variáveis

((a = 2 * b) > c)

// a = 2,  a comparação resulta em 1 se for verdadeiro ou 0 se for falso que multiplica o C
```

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

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

Regra 1 - Todo char e short int é convertido para int. Todo float é convertido para double.

Regra 2 - Para todos os pares de operandos, ocorre o seguinte, em seqüência: se um dos operandos for um long double, o outro será convertido para long double. Se um dos operandos for double, o outro será convertido para double. Se um dos operandos for long, o outro será convertido para long. Se um dos operandos for unsigned, o outro será convertido para unsigned.

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

(tipo) expressão

Exemplo:

```
main()

{

int x = 3;

printf("%f %f\n", (float) x / 2, (float) (x / 2));

/* serão impressos na tela 1.5 e 1.0 pois no primeiro caso, x é convertido para float e depois é dividido, já no segundo, somente o resultado é convertido para float */

}
```

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

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

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

```
/* Exemplo de variável static */

main()

{

        printf("%d\n", numero()); // imprimirá 0
```

```
        printf("%d\n", numero()); // imprimirá 1
}

numero()
{
        static valor = 0; // atribuição só executada 1 vez
        return valor ++;
}
```

printf() - Rotina de finalidade geral para saída pelo console n A função printf() serve para mostrar mensagens na tela. Sua forma geral é printf("string de controle", lista argumentos);

"string de controle" - A string de controle consiste em dois tipos de itens:

- Caracteres que a função imprimirá na tela.

- Comandos de formatação.

Comandos de formatação;

- Todos os comandos de formatação começam por % que é seguido pelo código de formatação

- Deve haver exatamente o mesmo número de argumentos quanto forem os comandos de formatação e eles devem coincidir em ordem.

Código	Formato
%c	um único caracter
%d	decimal
%i	Decimal iteger
%e	notação científica

%f	ponto decimal flutuante
%g	Usa %e ou %f - aquele que for menor
%o	Octal
%s	string de caracteres
%u	decimal sem sinal
%x	hexadecimal
%%	Sinal %
%p	exibe um ponteiro
%n	o argumento associado será um ponteiro inteiro no qual será colocado

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

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

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

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

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

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

Exemplos.

Comando printf()	**Resultado**
(“%-5.2f”, 123.234)	123.23
(“%5.2f”, 3.234)	3.23
(“%10s”, “alo”)	alo

(“%-10s”, “alo”)	alo
(“%5.7s”, “123456789”)	1234567
(“%5.4d”, 123)	0123

scanf() - Rotina de finalidade geral para entrada pelo console

A função scanf() serve para ler informações do teclado. Sua forma geral é scanf(“string de controle”, lista argumentos);

“string de controle” - A string de controle consiste em três classificações de caracteres:

Especificadores de formato

Caracteres brancos

Caracteres não-brancos

Especificadores de formato

Todos os especificadores de formato começam por % que é seguido por um caracter que indica o tipo de de dado que será lido.

Deve haver exatamente o mesmo número de argumentos quanto forem os especificadores de formato e eles devem coincidir em ordem.

Código	Lê
%c	único caracter
%d	decimal
%i	Decimal integer
%e	número com ponto flutuante
%f	numero com ponto decimal flutuante
%o	número octal
%s	string de caracteres
%x	hexadecimal
%p	ponteiro
%n	recebe um valor inteiro igual ao número de caracteres lidos até o momento

Caracter branco

- Um caracter branco na string de controle faz com que scanf() passe por cima de um ou mais caracteres brancos na string de entrada

- Um caracter branco é um espaço, um tab ou um \n

Caracter não branco

- Um caracter não branco na string de controle faz com que scanf() leia e desconsidere um caracter coincidente. Se o computador não encontrar o caracter especificado, scanf terminará.

- O comando “%d,%d” fará com que scanf leia um inteiro, depois leia e desconsidere uma vírgula e finalmente, leia um outro inteiro.

Como chamar scanf()

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

```
scanf(“%d”, &cont);
```

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

Ignorar entrada

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

```
scanf(“%d%*c%d”, &x, &y)
```

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

Comprimento máximo do campo

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

```
scanf(“%20s”, nome);
```

Espaços, tabs e \n

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

```
scanf(“%c%c%c”, &a, &b, &c);
```

seja lido com a entrada x y n scanf retornará com x em a, espaço em b e y em c.

Comandos de controle de fluxo

Os comandos de controle de fluxo são a base de qualquer linguagem.

A maneira como eles são implementados afeta a personalidade e percepção da linguagem.

C tem um conjunto muito rico e poderoso de comandos de controle de fluxo.

Eles se dividem em comandos de teste de condições e comandos de controle de loop.

Comandos de testes de condições

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

• if

• switch

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

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

onde a parte else é opcional. Por exemplo

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

if aninhados

C permite que sejam colocados comandos if dentro de outros comandos if. A isto chamamos de if aninhados.

Quando se trata de if aninhados, o comando else se refere ao if mais próximo que não possui um comando else. Tanto o if quanto o else devem estar dentro do mesmo bloco de código.

Aqui termina esta primeira parte introdutória.