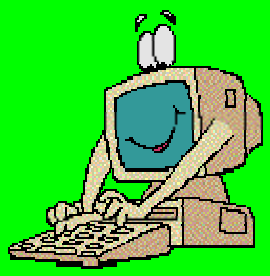
Universidade Estadual de Mato Grosso do Sul – UEMS
Bacharelado em Ciência da Computação



Linguagem de Programação C

Prof. Dr. Nilton Cézar de Paula
nilton@comp.uems.br

Dourados, março de 2009.

SUMÁRIO

1-Linguagem de Programação C
1.1 Introdução 01
1.2 Histórico 01
1.3 Principais características da linguagem 02
1.4 Palavras reservadas 02
1.5 Biblioteca de funções e linkedição 03
1.6 Anatomia de um programa 03
1.7 Usando o ambiente de desenvolvimento Borland Turbo C 04

2-Técnicas Gerais para Programação
2.1 Tipos de dados 06
2.2 Modificadores de tipo 06
2.3 Nome de identificador 07
2.4 Variáveis 07
2.5 Entrada e saída pelo console 09
2.6 Entrada e saída formatada 11
2.7 Constantes 13
2.8 Operadores 15
2.9 Expressões 19
2.10 Comandos de controle de programa 20
2.11 Modularização usando funções 29

3-Manipulação de Estruturas de Dados
3.1 Vetores e matrizes 32
3.2 Strings 33
3.3 Ponteiros 34
3.4 Alocação dinâmica de memória 37
3.5 Estruturas de dados heterogêneas 39

4-Manipulação de Arquivos
4.1 Entrada e saída com arquivos 42

5-Gráficos
5.1 Biblioteca gráfica da linguagem C 46
5.2 Rotinas de mouse integradas ao modo gráfico 49

Referências Bibliográficas 54

PREFÁCIO

Prezado leitor,

Este material é destinado aos alunos do Curso de Bacharelado em Ciência da Computação da Universidade de Mato Grosso do Sul – UEMS e também a todos os profissionais da área interessados em aprender ou aperfeiçoar seus conhecimentos em linguagens de programação de computadores, neste caso particular das linguagem C e C++.

Neste material são apresentados os conceitos básicos, estruturas e técnicas de programação C, que servirão como base para o estudo dos principais conceitos, fundamentos e técnicas da Programação Orientada a Objetos (*Object Oriented Programming*) utilizando a linguagem de programação C++.

É muito importante, para o estudo de linguagens de programação, que se faça uma leitura minuciosa dos textos que fundamentam a atividade de programação, reconhecidamente uma das mais árduas tarefas de um profissional de computação. Neste ponto, uma boa fundamentação teórica facilita a compreensão da sintaxe da linguagem e conduz a um bom aprendizado e, posteriormente, domínio da arte do desenvolvimento de software.

Este material apresenta aspectos teóricos e programas-exemplo que auxiliam a compreensão das técnicas de programação. Tabelas e figuras são também utilizadas na formação deste material didático. Em meio aos textos, são apresentados alguns ícones que devem ser observados com atenção. Tais ícones objetivam alertar o leitor para pontos de maior relevância do conteúdo deste material, bem como apresentar dicas e auxiliar o processo de aprendizagem.

Implemente o programa-exemplo para melhor compreensão do assunto

Espero que este material didático seja útil e colabore para a sua formação profissional.

O autor.

1 Linguagem de Programação C

1.1 Introdução

Existem várias linguagens de programação disponíveis no mercado de informática atualmente, cada qual com sua aplicação específica. Pode-se citar, por exemplo a linguagem PASCAL que tem como vantagem a facilidade de se aprender, e é assim muito utilizada por quem deseja iniciar o estudo de programação. Com uma aplicação diferente de PASCAL, pode-se citar a linguagem SQL (*Structured Query Language*), que é voltada para aplicações de banco de dados. Num outro ponto, existem linguagens como LISP e PROLOG, utilizadas em programação matemática e prova de teoremas, e para o desenvolvimento de sistemas inteligentes, respectivamente.

Por outro lado, existe a linguagem C, criada e implementada por Dennis Ritchie, como evolução do desenvolvimento de uma linguagem mais antiga chamada BCPL. Atualmente, existem várias implementações de C, mas para garantir compatibilidade entre estas implementações foi criado um padrão ANSI (*American National Standard Institute*) para a linguagem C, o que gerou a denominação C ANSI. Em adição, uma extensão da linguagem C, denominada C++, foi desenvolvida para acolher o paradigma da orientação a objetos. Por fim, em meados de 1995, surgiu a mais recente linguagem derivada da "família" C: a linguagem JAVA, que incorpora, dentre outras coisas, a orientação a objetos da linguagem C++, e estende as capacidades de uso em multiplataformas da linguagem C ANSI.

Não é interessante comparar todas estas linguagens entre si e dizer qual é a melhor. Estas linguagens são ferramentas para a tarefa de programação, onde cada uma com suas características próprias satisfaz a uma determinada aplicação a que se destina. Não existe uma linguagem que é “ótima” em qualquer situação; todas têm vantagens e desvantagens e sua utilização é definida pela aplicação e pela facilidade que o programador tem em operá-la.

1.2 Histórico

A linguagem C foi inventada e implementada por Dennis Ritchie, resultado do desenvolvimento de uma linguagem mais antiga chamada BCPL.

Linguagem BCPL → Linguagem B → Linguagem C

A linguagem C e o Sistema Operacional UNIX surgiram praticamente juntos e um ajudou o desenvolvimento do outro. A versão K&R (Brain Kernighan e Dennis Ritchie) já era fornecida junto com o Sistema Operacional UNIX versão 5 (System V).

Existem várias implementações de C, mas para garantir compatibilidade entre estas implementações foi criado um padrão ANSI (*American National Standard Institute*) para a linguagem C. O ANSI é um instituto americano que se encarrega da formulação de normas em diversos setores técnicos, incluindo os relativos aos computadores.

Atualmente, a linguagem C encontra-se estendida na versão orientada a objetos (usando a filosofia OOP), a linguagem C++ (inicialmente chamado C com classes). Esta versão continua vinculada ao padrão ANSI mas inclui ferramentas que facilitam o desenvolvimento de grandes sistemas usando a linguagem C. Mais recentemente, surgiu a linguagem JAVA que é derivada das linguagens C e C++.

1.3 Principais características da linguagem

A linguagem C é considerada de médio nível, ou seja, é uma linguagem que não está tão afastada da máquina.

Alto Nível	*Modula-2, Pascal, Cobol, JAVA, Visual Basic*
Médio Nível	*C, Macro-Assembler, C++*
Baixo Nível	*Assembler*

Tabela 01 - Classificação de algumas linguagens de programação

A linguagem C é portável, ou seja, um programa desenvolvido para uma máquina pode facilmente migrar para outra sem muitas alterações. O código gerado para um programa C é mais "resumido" (otimizado). A linguagem C tem somente 32 palavras-chave, sendo 27 do padrão K&R e 05 do padrão ANSI. Em adição, a linguagem C é estruturada, pois não admite declaração de função dentro de função. Além disso, admite malhas (laços) de repetição, indentação dos comandos e blocos de comandos (comandos compostos).

1.4 Palavras reservadas

Todas as palavras reservadas de C são escritas em caracteres minúsculos. A linguagem C faz distinção entre caracteres maiúsculos e minúsculos. Em todo programa C, as palavras reservadas podem ser utilizadas tanto na função **`main`**`()`, que é o corpo principal do programa, quanto nas funções implementadas pelo programador.

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

Tabela 02 - Palavras reservadas da linguagem C

1.5 Biblioteca de funções e linkedição

Além das palavras-chave, todo compilador C possui uma biblioteca padrão de funções que facilita a realização de tarefas como ler uma variável, comparar cadeia de caracteres, obter a data do sistema operacional e outras. Existe uma biblioteca básica definida pelo padrão ANSI, mas alguns compiladores C possuem algumas funções adicionais para aplicações gráficas, por exemplo.

As bibliotecas devem ser inseridas em um programa C que faça uso delas, e as funções usadas pelo programa são **`linkeditadas`** (ligadas) ao código executável do mesmo.

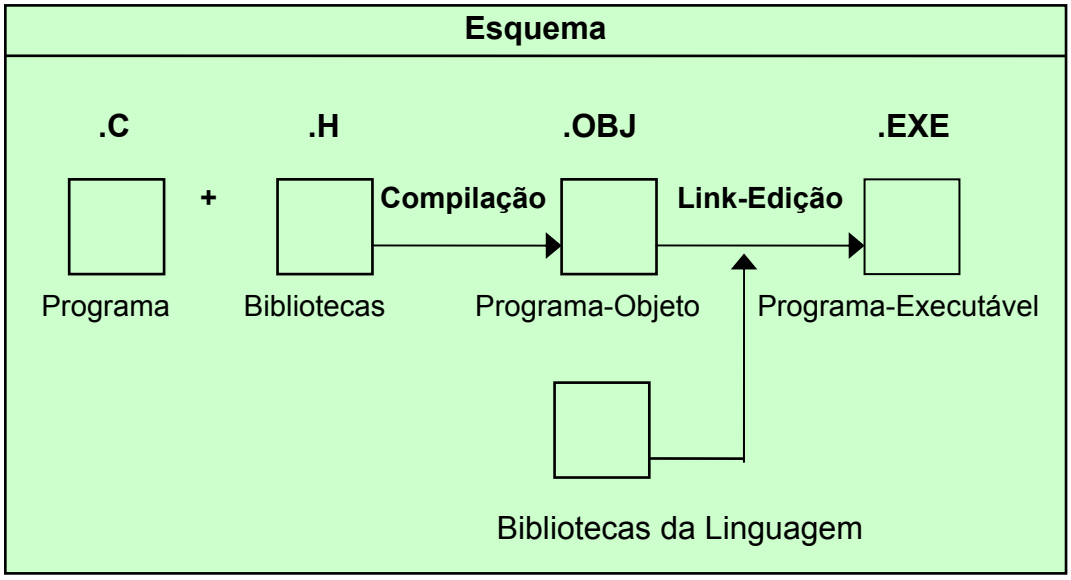


Figura 01 - Esquema de compilação da linguagem C

1.6 Anatomia de um programa

Todo programa em C segue uma estrutura básica em relação as partes que o compõem e que devem ser consideradas pelo programador, conforme ilustram a **Figura 02** e o **Programa 01**.

Anatomia de um Programa C

```
[inclusão de bibliotecas]
[declaração de dados globais]
[declaração dos protótipos de funções]

void main (void)
{
    [corpo do programa principal]
}

[implementação das funções do programa]
```

Figura 02 – Anatomia básica de um programa C

```
// Anatomia de um programa C

#include <stdio.h>

void main (void)
{
 puts ("Bem vindo a linguagem C");
}
```

Programa 01 – Anatomia básica de um programa C

1.7 Usando o ambiente de desenvolvimento Borland Turbo C

Os compiladores mais recentes da BORLAND (TURBO PASCAL, TURBO C, TURBO PROLOG e outros) estão embutidos em ambientes de programação (editor, compilador, linkeditor e depurador) de fácil de operação e muito poderosos. Tais ambientes são apresentados em forma de menus e janelas que se sobrepõem durante o desenvolvimento de um programa. Desta forma, é importante ressaltar as características principais dos componentes do ambiente de programação que fornecem suporte ao desenvolvimento de programas escritos na linguagem C.

O editor do TURBO C tem comandos equivalentes aos editores de texto genéricos, tais como o Microsoft Word. Além da edição, existem algumas operações por teclas de função e que auxiliam a programação.

F2	*Salva o arquivo corrente*
F3	*Lê/Carrega um arquivo do disco*
Alt-F3	*Apresenta os últimos oito arquivos que foram manipulados e que são apresentados numa pilha (no topo da pilha o arquivo mais recente)*
Alt-F6	*Troca, no editor, o último arquivo manipulado com o arquivo atual*
F10	*Volta ao menu principal do ambiente*

Tabela 03 - Algumas teclas importantes do ambiente Turbo C

O compilador e o linkeditor estão integrados, e para o programador parece ser uma única operação. Esta parte do ambiente de programação TURBO C se encontra num submenu do menu principal denominado **`Compile`**. Este submenu pode ser acessado de qualquer ponto do ambiente pela digitação das teclas **`ALT-C`**. As principais opções deste menu são:

Compile (Ctrl-F9)	*Compilar e Executar o programa*
Make (F9)	*Apenas Compilar o programa*
Build	*Refazer toda a Compilação do programa*

Tabela 04 - Funções principais do ambiente Turbo C

Por fim, uma das partes mais interessantes neste ambiente é sem dúvida o depurador (DEBUG). Com ele é possível acompanhar passo a passo a execução de um programa e identificar, facilmente, erros de lógica.

2 Técnicas Gerais para Programação

2.1 Tipos de dados

Em linguagens de programação o tipo de dados de uma variável define o conjunto de valores que a variável pode assumir. Por exemplo, uma variável de um tipo lógico., tal como **`boolean`**, pode assumir o valor **`true`** ou **`false`**. De modo geral, uma declaração de variável em uma linguagem de programação especifica duas coisas:

- a quantidade de bytes que deve ser reservada para a variável (por exemplo, no caso de uma variável inteira, para garantir que pode-se representar o maior inteiro permitido);
- como o dado representado por esses bytes deve ser interpretado (por exemplo, uma cadeia de bits pode ser interpretado como um inteiro ou real).

Então, tipos de dados podem ser vistos como métodos para interpretar o conteúdo da memória do computador. Na linguagem C existem vários tipos de dados primitivos, conforme mostra a **Tabela 05**.

Tamanho	Tipo	Explicação
1 byte	`char`	*caractere*
2 bytes	`int`	*inteiro*
4 bytes	`float`	*ponto flutuante (real)*
8 bytes	`double`	*ponto flutuante de precisão dupla*
0 bytes	`void`	*ausência de tipo*

Tabela 05 - Tipos de dados e sua representação

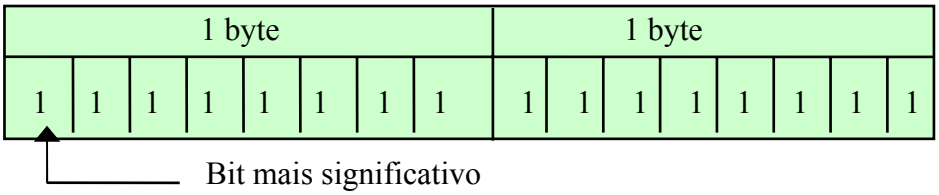
2.2 Modificadores de tipo

Um modificador de tipo é usado para alterar o significado de um tipo. Na linguagem C existem os seguintes modificadores de tipo: **`signed`**, **`short`**, **`long`**, **`unsigned`**.

Tipo	Tamanho em Bits	Faixa Máxima/Mínima
`char`	8	-128 a 127
`unsigned char`	8	0 a 255
`int`	16	-32768 a 32767
`unsigned int`	16	0 a 65535
`long int`	32	-2.147.483.648 a 2.147.483.647

Tabela 06 - Influência dos modificadores de tipo

A diferença entre inteiros com sinal (**signed**) e sem sinal (**unsigned**) é a maneira como o bit mais significativo é interpretado, conforme mostra a representação abaixo.



Neste contexto, se o número for um inteiro com sinal o valor máximo possível é 32767. Caso o número seja um inteiro sem sinal, o valor máximo será 65535.

2.3 Nome de identificador

Para declarar um identificador em C, deve-se observar as seguintes regras de formação de nomes de identificadores:

* o nome de um identificador deve começar com uma letra ou sublinhado (*underscore*) e os caracteres subseqüentes devem ser letras, número ou sublinhado;
* caracteres maiúsculos e minúsculos são tratados diferentemente;
* um identificador não pode ser igual a uma palavra-chave e não deve ser igual a nome de uma função em um programa.

Correto	Incorreto
`Contador`	`9contador`
`numero1`	`ola!`
`Caixa_um`	`caixa...um`
`_variavel`	`16`

Tabela 07 - Nomes corretos e incorretos para identificadores

2.4 Variáveis

Uma variável é uma posição de memória definida através de um nome (identificador) e que pode ter o seu valor modificado pela lógica do programa, durante sua execução. Como regra geral, observa-se que uma variável deve ser declarada antes de ser usada.

FORMA GERAL

```
tipo_de_dado  lista_de_variáveis;
```

As variáveis podem ser de 3 tipos, dependendo do local em que são declaradas:

* Variável local;
* Parâmetro formal;
* Variável global.

Uma variável é **local** quando só existe dentro do bloco no qual ela foi criada. Um bloco (comando composto) em C começa com **{** (abre chave) e termina com **}** (fecha chave).

```
// Apresenta a declaração de variáveis locais
void main (void)
{
  int valor;
}
```

Programa 02 – Declaração de variáveis locais

Uma variável é um **parâmetro formal** quando é passada para uma função que usa parâmetros. Os parâmetros formais são tratados como variáveis locais à função e sua declaração é feita depois do nome da função e dentro dos parênteses.

```
// Apresenta declaração de variáveis como parâmetro formal

#include <stdio.h>

void EscreveCaracter (char Caractere);

void main (void)
{
  EscreveCaracter('a');
}

void EscreveCaracter (char Caractere)
{
 putchar (Caractere);
}
```

Programa 03 – Declaração de variáveis como parâmetro formal

Se o tipo do parâmetro formal for diferente do tipo do argumento passado para a função, o programa em C não vai parar, mas isto pode incorrer em ERRO. O compilador C não verifica o número de argumentos passados para a função.

Uma variável é **global** quando está disponível ao programa inteiro; qualquer bloco do programa pode acessá-las sem erro. Estas variáveis devem ser declaradas fora de todas as funções e antes do início do programa principal.

```
// Apresenta declaração de variáveis globais

#include <stdio.h>
char Caractere;
void main (void)
{
  Caractere = 'a';
  putchar (Caractere);
}
```

Programa 04 – Declaração de variáveis globais

O uso de variáveis globais deve ser evitado porque estas ocupam espaço na memória durante toda o tempo de execução do programa. Deve ser explorado o uso de funções parametrizadas para melhorar a organização e a estruturação de programas em C.

A inicialização de variáveis em C é feita, simplesmente, pela atribuição de um valor à variável declarada.

FORMA GERAL

```
tipo_de_dado nome_da_variável = valor;
```

```
// Apresenta formas de inicialização de variáveis

#include <stdio.h>

void main (void)
{
 char Caractere = 'a';
 int Primeiro = 0;
 float Salario = 120.96;
}
```

Programa 05 – Inicialização de variáveis em um programa C

2.5 Entrada e saída pelo console

Existe uma série de funções de entrada e saída nas bibliotecas da linguagem C. No entanto, o enfoque das operações de entrada e saída é único, diferente de todas as linguagens de programação. Desta forma, existem funções na linguagem C para as seguintes operações:

* ler e escrever um caractere: **`getchar()`** e **`putchar()`**;
* ler e escrever cadeias de caracteres (strings): **`gets()`** e **`puts()`**;
* entrada e saída formatada: **`scanf()`** e **`printf()`**.

A função **`getchar()`** espera até que seja digitado um caractere no teclado; então mostra o caractere na tela e sai do processo de entrada de dados.

Protótipo
```
int getchar(void);
```

A função **`getch()`** efetua a leitura de um caractere do teclado sem ecoar o caractere na tela.

Protótipo
```
int getch(void);
```

A função **`getche()`** tem a mesma função de **`getchar()`**, porém não efetua o armazenamento em `buffer`, ecoando o caractere na tela.

Protótipo
```
int getche(void);
```

A função **`putchar()`** escreve um caractere a partir da posição corrente do cursor na tela.

Protótipo
```
int putchar(int Caractere);
```

```
// Programa que lê caracteres e escreve em caixa invertida.

#include <stdio.h>
#include <ctype.h>
#include <conio.h>
void main (void)
{
 char Caractere;
 puts ("\nDigite '.' e <ENTER> para terminar");
 do
 {
  Caractere = getchar ();
  if (islower (Caractere))
     Caractere = toupper (Caractere);
  else
      Caractere = tolower (Caractere);
      putchar (Caractere);
  } while (Caractere != '.');
}
```

Programa 06 – Uso de funções de entrada e saída

A função **gets()** efetua a leitura de um **string** de caracteres digitado através do teclado até que seja digitado <ENTER>. O caractere da tecla <ENTER> não faz parte do **string**; no seu lugar é colocado o caractere de fim de cadeia ('**\0**').

Protótipo

```
char *gets(char *string);
```

A função **puts()** escreve um **string** na tela seguido por uma nova linha ('**\n**'). É uma função mais rápida que a função **printf()**, mas somente opera com argumentos **string** (cadeias de caracteres).

Protótipo

```
int puts(char *string);
```

```
/* Programa que lê um nome e escreve uma mensagem
    com o nome digitado. */

#include <stdio.h>
#include <conio.h>
void main (void)
{
 char Nome [20];
 puts ("\nEscreva seu nome:");  gets (Nome);
 puts ("Prezado Senhor(a),");  puts (Nome);
 getch ();
}
```

Programa 07 – Uso de funções de entrada e saída com strings

2.6 Entrada e saída formatada

A função **printf ("série de controle", listadeargumentos)** é responsável pela saída formatada de dados na tela, onde:

* **"série de controle"**: é uma série de caracteres e comandos de formatação de dados que devem ser impressos;
* **listadeargumentos**: variáveis e constantes que devem ser trocados pelos formatos correspondentes na série de controle.

De maneira análoga, a função **scanf("série_de_controle", lista_de_endereços_dos_argumentos)** é responsável pela leitura (entrada) formatada de dados na tela. Os formatos usados na série de controle da função **scanf** são semelhantes aos formatos usados pela função **printf**.

```
// Programa que demonstra as formatações básicas de output
#include <stdio.h>
#include <conio.h>
void main (void)
{
 printf ("\n----+----+");
 printf ("\n%-5.2f", 123.234);
 printf ("\n%5.2f", 3.234);
 printf ("\n%10s", "hello");
 printf ("\n%-10s", "hello");
 printf ("\n%5.7s", "1234567890");
 getch ();
}
```

Programa 08 – Demonstração de saídas de dados formatadas

A constante de barra invertida **'\n'** significa salto para uma nova linha (*new line*). O formato **"%-5.2f"** indica que o número em ponto flutuante (**f**) deve ser apresentado com no mínimo 5 dígitos, sendo 2 dígitos para a parte fracionária do número e deve ser justificado à esquerda. O formato **"%5.2f"** indica que o número em ponto flutuante (**f**) deve ser apresentado com no mínimo 5 dígitos, sendo 2 dígitos para a parte fracionária do número e deve ser justificado à direita. O formato **"%10s"** indica que o **string** (s) deve ser apresentado em 10 espaços justificado a direita. O formato **"%-10s"** indica a mesma coisa que o anterior, só que justificado à esquerda. O formato **"%5.7s"** indica que o **string** (s) deve ser apresentado com pelo menos 5 caracteres e não mais que 7 caracteres.

Existem, ainda, alguns outros códigos de controle para a função **printf**, conforme mostra a **Tabela 08.**

Código	Formato
%c	*caractere*
%d	*inteiro c/ sinal*
%i	*inteiro c/ sinal*
%f	*números reais*
%o	*octal*
%x	*hexadecimal*
%%	*caractere '%'*

Tabela 08 - Tipos de constantes de formatação

```
// Programa que apresenta a Tabela do Quadrado e
// Cubo de números

#include <stdio.h>
#include <conio.h>
void main (void)
{
 int Numero;
 printf ("\n\n %8s %8s %8s", "x", "x^2", "x^3");
 for (Numero = 1; Numero <= 10; Numero++)
     printf("\n %8d %8d %8d", Numero, Numero*Numero,
            Numero*Numero*Numero);
 getch ();
}
```

Programa 09 – Demonstração de saídas de dados formatadas

```
// Programa que demonstra a entrada formatada de dados
#include <stdio.h>
#include <conio.h>
void main (void)
{
 char Nome [20]; int Idade;
 printf ("\nQual e o seu nome: ");
 scanf ("%s", Nome);  printf ("Quantos anos voce tem : ");
 scanf ("%d", &Idade);
 printf ("%s, voce tem %d anos", Nome, Idade);
 getch ();
}
```

Programa 10 – Demonstração de saídas de dados formatadas

2.7 Constantes

Uma constante é um valor fixo que não pode ser alterado durante a execução de um programa. Uma constante pode ser de um dos quatro tipos primitivos de dados:

- **`caractere (char)`:** 'a', '%' - envolvidos por aspas simples;
- **`inteiro (int)`:** 10, -97 - números sem componente fracionário;
- **`real (float,double)`:** 11.12 - número com componente fracionário;
- **`strings`:** "UEMS" - caracteres envolvidos por aspas duplas.

Uma **constante hexadecimal** é um número na base numérica 16 (hexadecimal). Os dígitos nesta base numérica são 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F, ou seja, 16 dígitos.

Exemplos
1A = 26 em decimal
FF = 255 em decimal

Para especificar ao compilador C que uma constante é hexadecimal, deve-se usar o símbolo **0x** antes do número.

Uma **constante octal** é um número na base numérica 8 (octal). Os dígitos nesta base numérica são 0,1,2,3,4,5,6,7 ou seja, 8 dígitos.

Exemplos
11 = 9 em decimal
24 = 20 em decimal

Para especificar ao compilador C que uma constante é octal, deve-se usar símbolo **0** (zero) antes do número.

```
// Demonstra o uso de Constantes Hexadecimais e Octais

#include "stdio.h"
#include "conio.h"
void main (void)
{
 int ConstanteH = 0xff;
 int ConstanteO = 024;
 printf("HEXADECIMAL: %X = DECIMAL : %d\n",ConstanteH,
           ConstanteH);
 printf("OCTAL: %o = DECIMAL : %d\n",ConstanteO,ConstanteO);
 getch ();
}
```

Programa 11 – Demonstração de uso de constantes

Uma **constante de barra invertida** é utilizada para descrever alguns caracteres da tabela ASCII que não podem ser definidos usando as aspas simples. A **Tabela 09** apresenta a lista de constantes de barra invertida da linguagem C.

Código	Significado
\a	*beep de alerta*
\f	*alimentação de formulário*
\n	*salto de linha*
\"	*caractere "*
\'	*caractere '*
\0	*marcador de final de cadeia de caracteres*
\\	*caractere *
\b	*retorcesso (backspace)*

Tabela 09 - Constantes de barra invertida

2.8 Operadores

Na linguagem C, existem 5 níveis de operadores destinados às operações de atribuição, aritméticas, relacionais, lógicas e `bit a bit`. A linguagem C utiliza o sinal de igual (**=**) como operador de atribuição. O formato geral de uma atribuição em C é o seguinte:

```
nome_da_variável = expressão;
```

Uma observação relevante é que o operador de atribuição pode entrar em qualquer expressão válida escrita na linguagem C.

Outra particularidade é a **`conversão automática de tipos`**. O valor do lado direito da atribuição é convertido no tipo do lado esquerdo. Se o tamanho em **`bytes`** do tipo do lado esquerdo for menor do que o valor atribuído, alguma informação é, então, perdida (os **`bits`** mais significativos do valor são desprezados).

```
// Programa que demonstra atribuições simples em C
 #include <stdio.h>
 #include <conio.h>
 void main (void)
 {
  int   X  = 10;
  char  Ch = 'a';
  float F  = 12.5;
  Ch = X;  // Os bits mais significativos de X são ignorados
  X = F;
  F = Ch;
  F = X;
 }
```

Programa 12 – Demonstração de operações de atribuição

As operações aritméticas em C são realizadas através do uso de operadores aritméticos encontrados na maioria das linguagens de programação, conforme mostra a **Tabela 10**.

Operador	Ação
-	*subtração*
+	*adição*
*	*multiplicação*
/	*divisão*
%	*módulo de divisão (resto da divisão)*
++	*incremento*
--	*decremento*

Tabela 10 - Operadores aritméticos

O uso dos operadores aritméticos é igual ao uso em outras linguagens, exceto os de incremento e decremento. Os operadores de incremento (**++**) e decremento (**--**), têm algumas particularidades. As instruções **x = x + 1** e **++x**, são instruções equivalentes em C, assim como, **x = x - 1** e **--x**. No entanto, os operadores de incremento e decremento podem preceder **(++x)** ou suceder **(x++)** o operando. Desta forma, existe uma diferença básica entre este dois usos. Pela instrução **++x** ou **--x**, o compilador C executa a operação de incremento ou decremento antes de usar o valor do operando. Já pela instrução **x++** ou **x--**, o compilador C executa a operação de incremento ou decremento depois de usar o valor do operando.

```
// Demonstra o uso do Operador de Incremento
 #include <stdio.h>
 #include <conio.h>
 void main (void)
 {
  int Numero1, Numero2;
  Numero1 = 10;  Numero2 = ++Numero1;
  printf("Numero1= %d Numero2= %d\n",Numero1,Numero2);
  Numero1 = 10;  Numero2 = Numero1++;
  printf("Numero1= %d Numero2= %d\n",Numero1,Numero2);
  getch ();
 }
```

Programa 13 – Uso de operadores de incremento e decremento

Para resolução de expressões aritméticas, é importante observar o fator de **precedência dos operadores**, conforme a ordem ilustrada pela representação seguinte.

Precedência mais alta

++ -- (incremento, decremento)
* / % (multiplicação, divisão, módulo)
+ - (adição, subtração)

Precedência mais baixa

Para alterar a ordem de precedência em uma expressão, deve-se usar o agrupamento, que é representado pelos parênteses.

Os operadores relacionais tratam das relações entre valores, e os operadores lógicos fazem a composição de expressões relacionais. Na linguagem C existe uma particularidade bastante interessante. Um valor é **VERDADEIRO** se `for` **diferente de zero**, e é **FALSO** se for **igual a zero**. Com relação aos operadores lógicos, observa-se a seguinte descrição e as operações contidas na **Tabela 11**.

&& (**AND** lógico) Ex.: a && b

|| (**OR** lógico) Ex.: a || b

! (**NOT** negação) Ex.: ! a

Operando	Operando	Expressão	Expressão	Operação
A	B	A && B	A || B	!A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Tabela 11 - Operadores lógicos e sua tabela verdade

```
// Programa que demonstra o uso dos Operadores lógicos
#include <stdio.h>
#include <conio.h>
void main (void)
{
 int N1 = 1, N2 = 0, N3;  N3 = N1 && !N2 || !N1 && N2;
 if  (N3)
    puts ("1");
 else
     puts ("0");
 getch ();
}
```

Programa 14 – Uso de operadores lógivos

A precedência de operadores é um fator importante para a resolução de expressões que resultam em valores lógicos. Assim, s linguagem C aplica a seguinte ordem de precedência para operadores lógicos e relacionais:

Precedência mais Alta

`!`	(negação)
`>, >=, <, <=, ==, !=`	(maior, maior ou igual, menor, menor ou igual, igual, diferente).
`&&`	(e lógico)
`||`	(ou lógico)

Precedência mais baixa

Todo resultado de uma expressão lógica e/ou relacional é igual a zero (**FALSO**) ou diferente de zero (**VERDADEIRO**).

```
// Programa que apresenta resultado de uma
// operação relacional

#include <stdio.h>
#include <conio.h>
void main (void)
{
 int X = 1;
 printf ("%d", X > 10);
 getch ();
}
```

Programa 15 – Complemento para uso de operadores relacionais

Finalizando, a linguagem C possui os operadores `bit` a `bit` que possibilitam a manipulação dos `bits` em um `byte`. As operações `bit` a `bit` são **AND**, **OR**, **XOR**, complemento, deslocamento à esquerda e deslocamento à direita. Essas operações atuam nos `bits` dos operandos.

Operador	Ação
&	*AND binário*
|	*OR binário*
^	*XOR binário*
~	*complemento*
>>	*deslocamento à direita*
<<	*deslocamento à esquerda*

Tabela 12 - Operadores bit a bit

Os operadores bit a bit são muito usados em programas *drivers* de dispositivos (modem, scanner, impressora, porta serial) para mascarar bits de informação nos bytes enviados/recebidos por estes dispositivos.

```
// Programa que apresenta tradução de hexadecimal
// para binário
#include <stdio.h>
#include <conio.h>
void main (void)
{
 int Hexadecimal = 0x7A, Indice, Mascara = 0x80;
 for (Indce = 1; Indice <= 8; Indice++)
 {
  if (Mascara & Hexadecimal)
     putchar ('1');
  else
      putchar ('0');
  Mascara = Mascara >> 1;
 }  getch ();
}
```

Programa 16 – Uso dos operadores bit a bit

Observa-se que o número hexadecimal 7A foi analisado pelo programa para ser traduzido para o seu equivalente binário. Operações bit a bit foram necessárias para obter os bits acesos (1) e apagados (0) do byte do valor hexadecimal

2.9 Expressões

Uma expressão é a composição de operadores lógicos, aritméticos e relacionais, variáveis e constantes. Em C uma expressão é qualquer combinação válida destes elementos. O compilador C converte todos os operandos no tipo do maior operando (promoção de tipo), conforme ilustra a **Figura 03**.

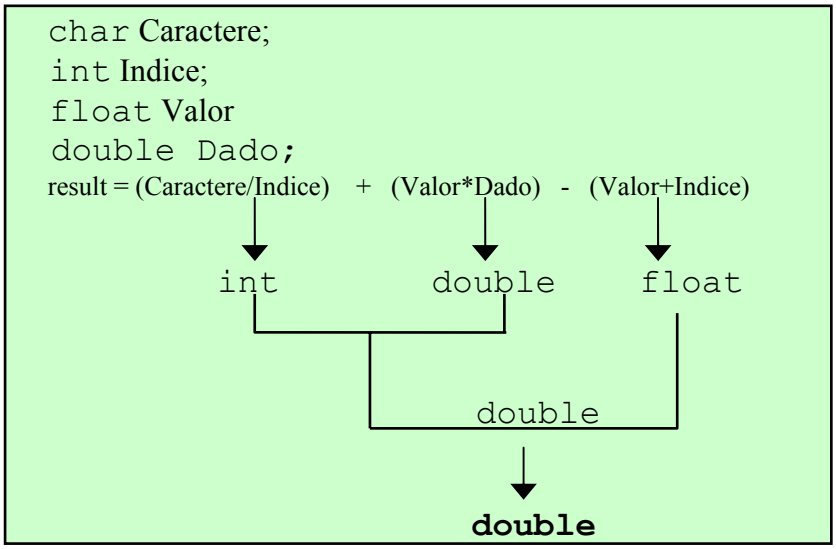


Figura 03 - Tipos e sua influência na resolução de expressões

Pode-se fazer com que o resultado de uma expressão seja de um tipo específico através do uso de um molde (*casts*) de tipo.

```
(float) Numero / 2;
```

Desta forma o resultado da variável **Numero** dividido por 2, é um valor real, independentemente do tipo original da variável Numero.

Espaços e parênteses podem ser usados em expressões na linguagem C sem que isto implique em diminuição de velocidade de execução da expressão. O uso de parênteses e espaços é aconselhado no sentido de aumentar a legibilidade do programa. Em adição, os parênteses podem mudar a ordem de precedência dos operadores.

Outra particularidade da linguagem C, refere-se ao uso de abreviações. A linguagem C admite que algumas atribuições sejam abreviadas.

```
X = X + 10;
```

Eqüivale à expressão

```
X += 10;
```

2.10 Comandos de controle de programa

O padrão ANSI divide os comandos em C nos seguintes grupos:

* **seleção** : `if` , `switch`;
* **iteração** : `while`, `do-while`, `for`;
* **desvio** : `break`, `continue`, `goto` e `return`;
* **expressão** : expressões válidas em C;
* **bloco** : blocos de código entre { e }.

Os comandos de seleção são empregados na construção de estruturas condicionais que indicam situações em que se deve tomar decisões na lógica de um programa.

Forma Geral

```
if (expressão)
   comando1;
else
    comando2;
```

A expressão que controla a seleção deve ser colocada entre parênteses. O **comando1**, que pode ser um comando simples ou um comando em bloco, será executado se o resultado da expressão for diferente de zero (VERDADEIRO). Por

outro lado, o **comando2** será executado se o resultado da expressão for igual a zero (FALSO).

Diferentemente de PASCAL, em C é necessário inserir um ponto e vírgula antes da palavra reservada **else**.

```
// Programa que apresenta o comando de seleção simples - if

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <conio.h>
void main (void)
{
 int Numero, Palpite;
 randomize ();
 Numero = rand ();
 puts ("\nAdivinhe o numero magico");
 scanf ("%d", &Palpite);
 if (Palpite == Numero)
    puts ("\n Voce ACERTOU...!");
 else
     puts ("\n Voce errou...!");
 getch ();
}
```

Programa 17 – Uso da estrutura de seleção simples

Na linguagem C, assim como em outras linguagens de programação, pode-se ter vários comandos **if** aninhados. As regras para aninhamento de comandos **if** é igual as encontradas em outras linguagens de programação. O comando **else** em C refere-se ao **if** mais próximo (acima deste) e que esteja no mesmo bloco de comandos.

Forma Geral

```
if (i)
{
 if (j)
   comando1;
 if (k)
   comando2;
else
   comando3;
}
else
   comando4;
```

```
// Programa que demonstra condições de seleção

#include <stdio.h>
#include <conio.h>

void main (void)
{
 int N1, N2;
 puts ("\nDigite dois numeros separados por espaco : ");
 scanf ("%d %d", &N1, &N2);
 if  (N2)
    printf ("\nDivisao : %d\n", N1/N2);
 else
     printf ("\nErro: Divisao por Zero\n");
 getch ();
}
```

Programa 18 – Uso da estrutura de seleção simples com aninhamento

Os comandos de seleção composta são empregados na construção de estruturas condicionais múltiplas e que, dentre as opções de seleção, apenas uma é escolhida para execução.

Forma Geral

```
switch (expressão)
{
 case constante1 : sequência_de_comandos1;
                   break;
 case constante2 : seqüência_de_comandos2;
                   break;
...
 case constanteN : seqüência_de_comandosN;
                   break;
 default : seqüência_de_comandos;
}
```

Se o resultado da expressão for igual ao valor da `constante1`, é executada a `seqüência_de_comandos1`. Se o resultado da expressão for igual ao valor da `constante2`, é executada a `seqüência_de_comandos2`. A `seqüência_de_comandos` após a palavra-chave **`default`** é executada se o resultado da expressão não for igual a nenhuma das constantes. O comando **`break`**, após cada seqüncia de comandos, põe fim ao comando **`switch`**. Se o comando **`break`** não for usado, todas as instruções contidas abaixo da seqüência de comandos selecionada são executadas, de forma indesejada.

```
// Programa que apresenta comando de seleção múltipla

#include <stdio.h>
#include <conio.h>

void main (void)
{
 char Opcao;
 printf ("\nDigite um numero: ");
 Opcao = getche (); printf ("\n");
 switch (Opcao)
 {
  case '1' : printf ("um "); break;
  case '2' : printf ("dois "); break;
  case '3' : printf ("tres "); break;
 }
 getch ();
}
```

Programa 19 – Uso da estrutura de seleção múltipla

Os comandos de iteração (malha ou laço de repetição) são usados para permitir que um conjunto de instruções seja executado enquanto uma determinada condição for verdadeira. Os comandos de iteração em C são: **`for`**, **`while`**, **`do while`**.

Forma Geral

`for` (inicialização; condição; incremento)
comando;

A `inicialização` é geralmente um comando de atribuição que coloca um valor inicial para as variáveis de controle do laço **`for`**. A `condição` é uma condição relacional, lógica ou aritmética, que define até quando a iteração pode continuar (condição de permanência da repetição). O `incremento` define como a variável de controle do laço varia cada vez que o laço é repetido.

```
// Programa que demonstra o uso da malha de repetição for
#include <stdio.h>
#include <conio.h>
void main (void)
{
 int Contador;
 for (Contador = 1; Contador < 100; Contador++)
      printf ("%-3d", Contador);
 getch ();
}
```

Programa 20 – Uso da estrutura de repetição *for*

```
// Malha de repetição for com comando composto
#include <stdio.h>
#include <conio.h>
void main (void)
{
 int Indice, Numero;
 for (Indice = 100; Indice != 65; Indice -= 5)
 {
  Numero = Indice * Indice;
  printf ("\nO quadrado de %3d e' %5d ", Indice, Numero);
 }
 getch ();
}
```

Programa 21 – Uso da estrutura de repetição *for*

O laço de repetição **`for`** pode trabalhar com mais do que uma única variável. Assim, pode-se inicializar e incrementar mais do que uma variável no escopo da repetição `for`.

```
// Repetição for com duas variáveis de controle
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main (void)
{
 char Nome[9] = "Dourados";  int Indice1,Indice2;
 clrscr ();
 for (Indice1=0, Indice2=strlen (Nome) - 1;
       Indice1 < strlen (Nome); Indice1++, Indice2--)
 {
  gotoxy (10+Indice1,1); putchar (Nome[Indice1]);
  gotoxy (10+Indice1,2); putchar (Nome[Indice2]);
 }
 getch ();
}
```

Programa 22 – Uso da estrutura de repetição *for*

Deve-se observar, ainda, que parte das definições do laço **`for`** não são obrigatórias e, com isto, consegue-se, por exemplo, um laço com repetição infinita.

```
// Repetição for infinita

#include <stdio.h>
void main (void)
{
 for (;;)
     printf ("Laco sem fim!");
}
```

Programa 23 – Uso da estrutura de repetição *for*

Existem também os casos em que uma malha de repetição não executa comandos. Neste ponto, a repetição é conhecida como laço sem corpo.

```
// Repetição for sem corpo

#include <stdio.h>
#include <conio.h>
void main (void)
{
 long int Contador;
 clrscr ();
 puts ("Espere um pouco...");
 for (Contador = 1L; Contador < 5000000L; Contador++);
 puts ("Terminei !");
 getch ();
}
```

Programa 24 – Uso da estrutura de repetição *for*

A estrutura de repetição `while`, assim como a estrutura `for`, permite que um conjunto de comandos possa ser executado enquanto uma determinada condição de permanência for verdadeira.

Forma Geral

`while` (condição)
comando;

A `condição` é qualquer expressão onde o resultado é um valor igual ou diferente de zero. O `comando` é um comando vazio, um comando simples ou um bloco de comandos (comando composto). Observa-se que a condição de parada é inserida no início da estrutura de repetição.

```
// Programa que apresenta a malha de repetição while

#include <stdio.h>
#include <conio.h>
void main (void)
{
 char Caractere = 0;
 while (Caractere != 'A')
       Caractere = getche ();
 puts ("\nFim");
 getch ();
}
```

Programa 25 – Uso da estrutura de repetição *while*

Ao contrário dos laços **while** e **for** que testam a condição de permanência no início da estrutura de repetição, o laço **do while** testa a condição de permanência no final, fazendo com que os comandos internos à estrutura de repetição sejam executados ao menos uma vez.

Forma Geral

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

Os comandos dentro das chaves são executados enquanto a condição de permanência for satisfeita (VERDADEIRA).

```
// Programa que apresenta a malha de repetição do-while
#include <stdio.h>
#include <conio.h>
void main (void)
{
 int Contador = 1;
 clrscr ();
 do
 {
  printf ("%4d", Contador);
  Contador++;
 } while (Contador <= 100);
 getch ();
}
```

Programa 26 – Uso da estrutura de repetição *do while*

O comando de desvio **return** obriga o programa a retornar da execução de uma função. Se o programa está executando uma função e encontra o comando **return**, ocorre um retorno ao programa principal exatamente após o ponto em que foi chamada a função.

Forma Geral

return expressão;

```
// Apresenta o comando de desvio return
#include <stdio.h>
#include <conio.h>
int Subtrai (int N1, int N2);
void main (void)
{
 int Numero1, Numero2;
 puts ("\nDigite dois inteiros separados por espaco : ");
 scanf ("%d%d", &Numero1, &Numero2);
 printf("%d menos %d e' igual a %d", Numero1, Numero2,
         Subtrai (Numero1,Numero2));  getch ();
}
int Subtrai (int N1, int N2)
{
 return (N1 - N2);
}
```

Programa 27 – Uso do comando de desvio *return*

O comando **break** tem dois usos: terminar uma estrutura condicional **case** da instrução **switch**, ou terminar uma malha de repetição evitando o teste de condição de permanência.

```
// Apresenta o uso do comando de desvio break
#include <stdio.h>
#include <conio.h>
void main (void)
{
 int Tempo = 1;
 clrscr ();
 for (;;)
 {
  if (Tempo > 100)
     break;
  printf ("%4d", Tempo++);
 }
 getch ();
}
```

Programa 28 – Uso do comando *break*

Assim como é possível interromper uma malha de repetição com a instrução **break**, pode-se também interromper a execução de um programa com a instrução **exit**.

Forma Geral
`void exit (int código_de_retorno);`

O valor do `código_de_retorno` é devolvido ao processo chamador do programa, que é normalmente o sistema operacional.

```
// Apresenta o uso da função de desvio exit

#include <stdlib.h>
void main (void)
{
 exit (1);
}
```

Programa 29 – Uso do comando *exit*

O comando **continue** tem efeito contrário ao do comando **break**; obriga a execução da próxima iteração do laço de repetição, evitando a execução de qualquer código intermediário. Para o laço **for**, o comando **continue** faz com que a condição e o incremento sejam executados.

```
// Apresenta o uso do comando de desvio continue
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main (void)
{
 char Nome[80];
 int Indice;
 printf ("\nDigite um nome completo: ");
 gets (Nome);
 for (Indice = 0; Indice != strlen (Nome); Indice++)
 {
  if (Nome [Indice] == ' ')
     continue;
  putchar (Nome[Indice]);
 }
 getche ();
}
```

Programa 30 – Uso do comando *continue*

2.11 Modularização usando funções

Uma função na linguagem C pode executar as mesmas tarefas que os procedimentos e funções de outras linguagens de programação estruturadas.

Forma Geral

```
tipo nome_da_função (lista_parâmetros)
{
 corpo da função
}
```

O **`tipo`** pode ser qualquer um dos tipos de dados válidos da linguagem C. Se não for definido nenhum tipo para função, a linguagem C assume que a função devolve um valor inteiro (`int`). O `nome_da_função` é qualquer nome válido de identificador em C pelo qual a função é identificada (chamada). A `lista_parâmetros` é uma lista de nomes de variáveis, separadas por vírgulas, e seus tipos associados. Estas variáveis recebem os valores dos argumentos passados na chamada da função.

```
// Apresenta a construção de uma função em C
#include <stdio.h>
#include <conio.h>
Contido (char Caractere, char *Cadeia);
void main (void)
{
 char Nome [10] = "Unifenas", Caractere = 'a';  clrscr ();
 if (Contido (Caractere, Nome))
    puts ("O caractere esta' no nome");
 else
     puts ("O caractere nao esta' no nome");
}
Contido (char Caractere, char *Cadeia)
{
 while (*Cadeia)
       if (*Cadeia == Caractere)
          return 1;
       else
           Cadeia++;
 return 0;
}
```

Programa 31 – Modularização através de funções

De modo geral, assim como em outras linguagens de programação, um função pode ser definida de três formas: sem parâmetros, com parâmetros por valor e com parâmetros por referência.

Uma função com parâmetros por valor é caracterizada pelo uso normal dos parâmetros formais da função sem alterar os valores dos argumentos. Neste caso, a função "funciona como função".

```
// Apresenta função com parâmetros por  valor

#include <stdio.h>
#include <conio.h>
int Quadrado (int Numero);
void main (void)
{
 int N1 = 8;
 printf ("\nO quadrado de %d e' %d", N1, Quadrado (N1));
 getch ();
}

int Quadrado (int Numero)
{
 return Numero * Numero;
}
```

Programa 31 – Função com parâmetros por valor

Na implementação de funções com parâmetros por referência, o valor do argumento passado para a função é alterado pelo corpo da função e devolvido ao programa principal ou outra função (ponto que chamou a função). Neste caso, a função "funciona como procedimento".

```
// Apresenta função com parâmetros por referência
#include <stdio.h>
#include <conio.h>
void Quadrado (int Numero, int *Resultado);
void main (void)
{
 int N1 = 8, Resultado;
 Quadrado (N1, &Resultado);
 printf ("\nO quadrado de %d e' %d", N1, Resultado);
 getch ();
}

void Quadrado (int Numero, int *Resultado)
{
 *Resultado = Numero * Numero;
}
```

Programa 32 – Função com parâmetros por referência

Na passagem por referência, deve-se passar o endereço do argumento que se deseja alterar dentro da função. Nas bibliotecas de C, existem funções que usam este mecanismo. Um exemplo prático é a função **`scanf().`**

```
int Numero;
scanf ("%d", &Numero);
```

Complementando, várias funções de biblioteca C podem ser incluídas em programas. Isto é feito com a inclusão de um arquivo de cabeçalho no início do programa. Este arquivo de cabeçalho (com extensão `.h`) contém uma série de definições dos protótipos de funções e de constantes. Por exemplo, com a inclusão do arquivo de cabeçalho **`STDIO.H`** (`#include "stdio.h"`), as funções de biblioteca, tais como **`printf`**, **`scanf`** e outras, tornam-se disponíveis ao programa. A seguir, é apresentada uma relação de alguns dos arquivos de cabeçalho e suas funções principais.

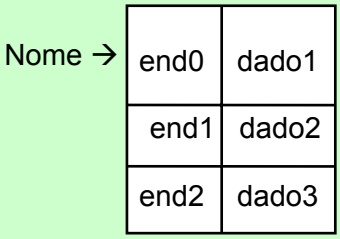
- **CONIO.H**
  `clrscr (); gotoxy (); kbhit (); getch ();`
- **STDIO.H**
  `gets (); puts (); printf (); scanf ();`
  `getchar ();`
- **STDLIB.H**
  `atof (); exit (); atol (); atoi ();`
- **STRING.H**
  `strcmp (); strcat (); strcpy ();`
  `strchr(); strlen ();`
- **GRAPHICS.H**
  `initgraph (); rectangle (); closegraph ();`
  `getimage(); putimage (); line ():`
- **DOS.H**
  `delay (); gettime (); setdate ();`
  `getdate(); sound ();`
- **CTYPE.H**
  `toupper (); tolower ();`

3 Manipulação de Estruturas de Dados

3.1 Vetores e matrizes

Uma matriz é uma coleção homogênea de dados que ocupam posições contíguas na memória e podem ser acessadas através de um índice. O endereço mais baixo na memória corresponde ao primeiro elemento da matriz e o mais alto corresponde ao último elemento da matriz.

Exemplo: `char` Nome [3];



O nome da matriz é um ponteiro para a primeira posição na memória onde está alocada a matriz. No exemplo, **`Nome`** é um ponteiro implícito para o primeiro elemento da matriz na memória. Neste ponto, uma observação relevante é que todas as matrizes em C começam no índice 0 (zero) e podem ser unidimensionais ou multidimensionais.

Uma matriz unidimensional, também conhecida por vetor, é uma estrutura de dados primitiva que permite a alocação de posições contíguas de memória, todas de um mesmo tipo de dados, e tais posições são acessadas pelo uso de um único índice que aponta para as posições de memória alocadas pela matriz.

Forma Geral de declaração

`tipo` nome_de_identificador `[` tamanho `];`

```
// Programa que demonstra o uso de vetores
#include <stdio.h>
#include <conio.h>
void main (void)
{
 int Indice, Vetor[100];
 for (Indice = 100; Indice > 0; Indice--)
       Vetor [Indice-1] = Indice;
 for (Indice = 1; Indice <= 100; Indice++)
       printf ("%4d", Vetor [Indice-1]);
}
```

Programa 33 – Demonstra o uso de vetores

Outro ponto importante é o fato de que a linguagem C não verifica limites em uma matriz. Desta forma, para o exemplo anterior a instrução `Vetor[2200] = 20` seria executada sem erro; no entanto, o programa teria um erro de lógica, pois outra posição de memória seria afetada de forma indesejável.

As matrizes bidimensionais, na verdade, comportam-se como vetores na memória, possuindo uma combinação linear de dois índices de acesso.

Forma geral

```
tipo nome_de_identificador [total_linhas][total_colunas];
```

```
// Programa que demonstra o uso de matrizes bidimensionais

#include <stdio.h>
#include <conio.h>

void main (void)
{
 int IndiceL, IndiceC, Matriz[3][4];
 for (IndiceL = 0; IndiceL < 3; IndiceL++)
     for (IndiceC = 0; IndiceC < 4; IndiceC++)
         Matriz [IndiceL][IndiceC] = IndiceL * IndiceC;
 for (IndiceL = 0; IndiceL < 3; IndiceL++)
     for (IndiceC = 0; IndiceC < 4; IndiceC++)
         printf ("%4d", Matriz [IndiceL][IndiceC]);
 getch ();
}
```

Programa 34 – Demonstra o uso de matrizes

3.2 Strings

Uma **`string`** em C é uma matriz de caracteres terminada com um caractere marcador de final de cadeia `'\0'` (o caractere nulo). Por isto, deve-se declarar uma **`string`** com um caractere a mais do que o necessário à aplicação. Assim, se é necessário uma **`string`** de 10 posições, deve-se declará-la com 11 posições.

```
char Cadeia[11]; // índices de (0 - 10)
```

Para manipulação de **`strings`** existe uma biblioteca C como uma série de funções, conforme apresenta a **Tabela 13**.

Função	Ação
strcpy(S1,S2)	*Copia a string S1 em S2*
strcat(S1,S2)	*Concatena S2 ao final de S1*
strlen(S1)	*Retorna o tamanho da string S1*
strcmp(S1,S2)	*Compara a string S1 com S2*

Tabela 13 - Algumas funções para manipulação de strings

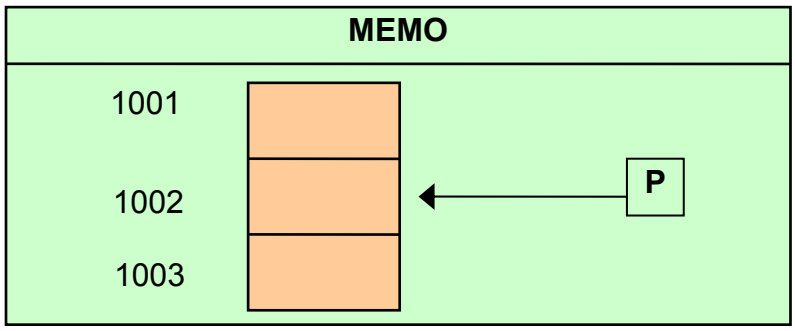
```
// Demonstra o uso das funções para strings em C
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main (void)
{
 char Nome1 [40], Nome2 [40];
 clrscr ();
 printf ("\nDigite um nome    : ");  gets (Nome1);
 printf ("Digite outro nome : ");  gets (Nome2);
 puts ("\n**** Teste da Funcao STRLEN     ");
 printf ("Tamanho do primeiro nome : %d", strlen (Nome1));
 printf ("\nTamanho do segundo  nome : %d", strlen (Nome2));
 puts ("\n\n**** Teste da Funcao STRCMP     ");
 if (!strcmp (Nome1, Nome2))
    puts ("Os nomes sao iguais");
 else
     puts ("Os nomes sao diferentes");
 puts ("\n**** Teste da Funcao STRCAT     ");
 strcat (Nome1, Nome2);
 printf ("Nomes concatenados : %s ", Nome1);
 puts ("\n\n**** Teste da Funcao STRCPY     ");
 strcpy (Nome1, "Universidade de Alfenas");  getch ();
}
```

Programa 35 – Funções para manipulação de strings

Deve-se observar que a função **strcmp** possui particularidades importantes. Ela retorna o valor 0 (zero), se a `String1` e a `String2` são iguais; retorna um valor menor que 0, se a `String1` e é alfabeticamente menor que a `String2`; e retorna um valor maior que 0, se a `String1` é alfabeticamente maior que a `String2`.

3.3 Ponteiros

Os ponteiros são variáveis que apontam para algum endereço de memória. São usados para alocação dinâmica de espaço em memória, ou seja, alocação de espaço de memória que ocorre durante o tempo de execução dos programas (*run time*).



No esquema acima, **P** é um ponteiro e aponta para o endereço 1002 da memória. De forma geral, um ponteiro pode ser declarado da seguinte maneira:

```
tipo * nome_da_variável;
```

Conforme a regra acima, uma declaração como **`char *Cadeia`**, cria um ponteiro para uma posição de memória cujo valor é caractere.

De modo geral, na linguagem C existem dois operadores que permitem a manipulação dos ponteiros:

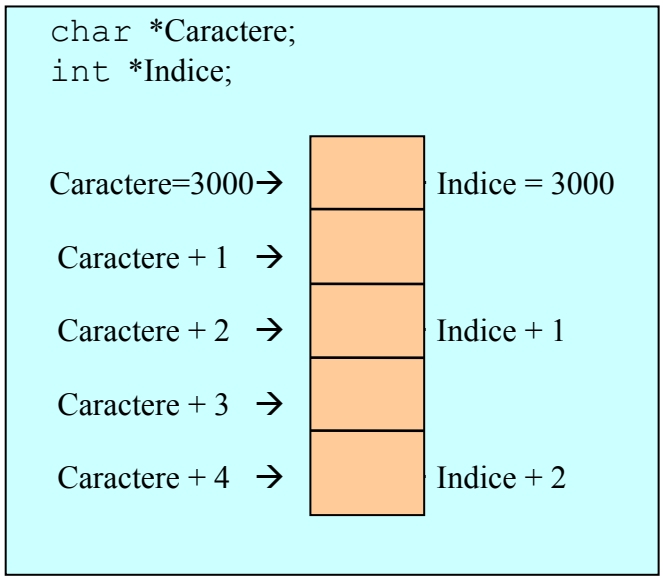
* **&** : devolve o endereço de uma variável (ponteiro);
* ***** : devolve o conteúdo de memória apontado por um ponteiro.

Usando ponteiros, pode-se efetuar operações como atribuir endereços e conteúdos, processar cálculos aritméticos com os conteúdos através de seus endereços ou ,ainda, efetuar comparações entre endereços e conteúdos.

```
// Programa que demonstra atribuições com ponteiros
#include <stdio.h>
#include <conio.h>
void main (void)
{
 int N1;
 int *Ponteiro1, *Ponteiro2;
 Ponteiro1 = &N1;
 Ponteiro2 = Ponteiro1;
 printf ("%p", Ponteiro2);
}
```

Programa 36 – Manipulação básica de ponteiros

Com relação ao processo de cálculo aritmético, duas operações são válidas para ponteiros: adição e subtração.



Quando um ponteiro é incrementado, na verdade está se efetuando um salto do tamanho do tipo de dados base na memória do computador. Na ilustração anterior, o incremento de 1 na variável **`Caractere`** provoca um salto para o próximo `byte` na memória (`Caractere` aponta para **`char`** que ocupa um `byte` na memória). O incremento de 1 na variável **`Indice`**, provoca um salto de dois `bytes` à frente na memória (o tipo base **`int`** ocupa dois `bytes` na memória).

Uma boa aplicação para evidenciar o trabalho com ponteiros, reside na manipulação de pilha usando vetor. Duas operações são válidas com a pilha: **`empilhar`** e **`desempilhar`** elementos do topo da pilha. No exemplo, a função **`Empilha`** é chamada toda vez que um valor diferente de zero é fornecido pelo usuário. A função **`Desempilha`** é chamada quando se entra com o valor zero. O valor -1 termina a execução do programa.

```
// Aplicação com ponteiros - Pilha usando vetor

#include <stdio.h>
#include <conio.h>

void Empilha (int Numero);
int Desempilha (void);

int *Tamanho, *Topo, Pilha[10];

void main (void)
{
 int Valor;
 clrscr ();
```

```
 puts ("\nEmpilha   : valor diferente de zero");
 puts ("Desempilha: valor igual a zero");
 puts ("Fim  : valor igual a -1");
 Tamanho = Pilha + 10; Topo = Pilha - 1;
 do
 {
  printf ("Digite um valor : ");  scanf ("%d", &Valor);
  if (Valor != 0)
     Empilha (Valor);
  else
      printf ("Desempilhei : %d\n", Desempilha());
 } while (Valor != -1);
}

void Empilha (int Numero)
{
 if (Topo + 1 == Tamanho)
 {
  puts ("Pilha cheia");
  return;
 }
 Topo++;
 *Topo = Numero;
}

int Desempilha (void)
{
 if (Topo == Pilha - 1)
 {
  puts ("Pilha vazia");
  return 0;
 }
 Topo--;
 return *(Topo + 1);
}
```

Programa 37 – Aplicação básica de ponteiros em pilhas

3.5 Alocação dinâmica de memória

Existe em C, uma forma poderosa de se gerar espaço de memória dinamicamente, durante a execução de um programa. Isto é necessário quando existem, nos programas, posições de memória que não podem ser definidas estaticamente, como por exemplo uma pilha dentro de um programa qualquer. A pilha deve crescer dinamicamente, conforme os elementos são gerados e empilhados. Utilizando o vetor como mecanismo para manutenção dessa pilha, estar-se-á limitando o número de elementos que podem ser mantidos na pilha. Com o uso da alocação dinâmica para os elementos da pilha, a única limitação é o espaço de memória disponível quando da execução do programa.

Basicamente, existem duas funções para alocação dinâmica de memória em C:

- **`malloc`**: devolve o endereço do primeiro `byte` da memória alocado para uma variável ponteiro;
- **`free`**: é a função simétrica de **`malloc`**, visto que ela devolve para o sistema uma porção de memória que foi alocada dinamicamente.

```
char *Ponteiro;
Ponteiro = malloc (1000);
```

Com a instrução acima, foram alocados com `1000 bytes` de memória sendo que a variável `Ponteiro` aponta para o primeiro desses `1000 bytes`. Desta forma, uma `string` foi alocada dinamicamente. Sempre que se alocar memória dinamicamente, deve-se testar o valor devolvido por **`malloc(),`** antes de usar o ponteiro, para garantir que não é nulo. Tal teste deve ser feito da seguinte forma:

```
if (!Ponteiro = malloc (1000))
{
 puts ("sem memoria\n");
 exit (1);
}
```

```
// Programa que mantem uma arvore binaria / alocacao
// dinamica

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>

struct Arvore
{
 char Informacao[30];
 struct Arvore *Direita;
 struct Arvore *Esquerda;
};

struct Arvore *Constroi (void);
void PreOrdem (struct Arvore *No);
void Destroi (struct Arvore *No);
void main ()
{
 struct Arvore *ListaNomes = NULL;
 ListaNomes = Constroi ();
 PreOrdem (&ListaNomes);
 Destroi (&ListaNomes);
}
```

```
struct Arvore *Constroi (void)
{
 struct Arvore *No;  char Auxiliar [30];
 gets (Auxiliar);
 if (strcmp (Auxiliar,".") == 0)
    return NULL;
 else
 {
  No = malloc (sizeof (struct Arvore));
  strcpy (No->Informacao, Auxiliar);
  No->Esquerda = Constroi ();
  No->Direita = Constroi ();
  return No;
 }
}

void PreOrdem (struct Arvore *No)
{
 if (No)
 {
  puts (No->Informacao);
  PreOrdem (No->Esquerda);
  PreOrdem (No->Direita);
 }
}

void Destroi (struct Arvore *No)
{
 if (No)
 {
  Destroi (No->Esquerda);
  Destroi (No->Direita);
  free (No); No = NULL;
 }
}
```

Programa 38 – Manipulação de alocação dinâmica de memória

3.5 Estruturas de dados heterogêneas

Estruturas são conjuntos heterogêneos de dados que são agrupados e referenciados através de um mesmo nome, sendo equivalente ao `REGISTRO` de outras linguagens de programação.

Forma Geral de Definição

```
struct nome_de_identificador
{
 tipo nome_da_variável;
 ...
 tipo nome_da_variável;
} variáveis_do_tipo_estrutura;
```

Para referenciar um elemento de uma estrutura deve-se utilizar o operador **ponto** (**.**). O nome da variável estrutura seguido por um ponto e pelo nome do elemento referencia o elemento individual da estrutura.

Uma estrutura inteira pode ser atribuída à outra, nas versões de C que são compatíveis com o padrão ANSI.

```
// Programa que apresenta atribuições de estruturas
#include <stdio.h>
#include <conio.h>
void main (void)
{
 struct Registro
 {
   int N1;
   float N2;
 };
 struct Registro Estrutura1, Estrutura2;
 Estrutura1.N1 = 7;
 Estrutura1.N2 = 7.5;
 Estrutura2 = Estrutura1;
 printf ("\n%d", Estrutura2.N1);
 getch ();
}
```

Programa 39 – Definição e uso básico de estruturas heterogêneas

A linguagem C permite a criação de estruturas de dados híbridas. Assim, pode-se declarar matrizes de estruturas heterogêneas em C. Por exemplo, para a estrutura de dados **Registro** declarada anteriormente, a seguinte matriz unidimensional é válida:

```
struct Registro Vetor[100];
```

Com esta declaração está sendo definida uma matriz de 100 posições, onde cada posição é uma estrutura de dados heterogênea **Registro**. Para se acessar uma estrutura específica, deve-se indexar o nome da estrutura. Por exemplo, se for necessário imprimir o campo **N1** da estrutura 3, deve-se codificar:

```
printf ("%d", Vetor[2].N1);
```

Assim como se faz para os ponteiros de tipos de dados primitivos, pode-se declarar ponteiros para estruturas de dados heterogêneas colocando o operador ***** antes do nome da estrutura.

```
struct Conta
{
 float Saldo;
 char Nome[30];
} Cliente;
struct Conta *Ponteiro;
Ponteiro = &Cliente;
```

A instrução acima coloca o endereço da estrutura de dados **Cliente** no ponteiro **Ponteiro**. Para acessar os campos de uma estrutura usando um ponteiro para estrutura, deve-se usar o operador **-> .**

```
Ponteiro -> N1
```

Como complemento, a linguagem C admite que se defina explicitamente novos nomes para os tipos de dados utilizando a palavra-chave **typedef**.

```
// Demonstra o uso de estruturas e funções com typedef
#include <stdio.h>
#include <conio.h>

typedef struct
{
 int N1, N2;
 char Caractere;
} Registro;

void AlteraValor (Registro *Dados);

void main (void)
{
 Registro Informacoes;
 Informacoes.N1 = 1000;
 AlteraValor (&Informacoes);
 printf ("\n%d", Informacoes.N1);
 getch ();
}

void AlteraValor (Registro *Dados)
{
 Dados->N1 = 200;
}
```

Programa 40 – Definição e uso básico de estruturas heterogêneas

4 Manipulação de Arquivos

4.1 Entrada e saída com arquivos

Existem dois conjuntos de funções de E/S com arquivos na linguagem C. Num primeiro ponto, tem-se a E/S ANSI (com *buffer* ou formatada) e, em termos mais específicos, tem-se a E/S UNIX (sem *buffer* ou não formatada). Uma ênfase maior é dada ao primeiro conjunto pela portabilidade deste sistema de entrada e saída com arquivos.

O sistema de arquivos na linguagem C é definido para manipular uma série de dispositivos, tais como terminais, acionadores de disco e outros. Estes dispositivos são vistos como arquivos lógicos em C, denominados `STREAM` (abstração do dispositivo). O dispositivo real é denominado `ARQUIVO` (impressora, disco, console e outros). Um **`STREAM`** é associado a um `ARQUIVO` por uma operação de abertura do arquivo e, a partir da associação, todas as demais operações de escrita e leitura podem ser realizadas. A **Tabela 14** apresenta as principais funções da linguagem C para manipulação de arquivos.

Função	Ação
`fopen()`	*Abre um arquivo*
`fclose`	*Fecha um arquivo*
`putc() e fputc()`	*Escreve um caractere em um arquivo*
`getc() e fgetc()`	*Lê um caractere de um arquivo*
`fseek()`	*Posiciona em um registro de um arquivo*
`fprintf()`	*Efetua impressão formatada em um arquivo*
`fscanf()`	*Efetua leitura formatada em um arquivo*
`feof()`	*Verifica o final de um arquivo*
`fwrite()`	*Escreve tipos maiores que 1 byte em um arquivo*
`fread()`	*Lê tipos maiores que 1 byte de um arquivo*

Tabela 14 - Funções do padrão ANSI para manipulação de arquivos

O ponteiro de arquivo une o sistema de E/S a um `buffer`. O ponteiro não aponta diretamente para o arquivo em disco, mas contém informações sobre o arquivo, incluindo nome, `status` (aberto, fechado e outros) e posição atual sobre o arquivo. Para definir uma variável ponteiro de arquivo deve-se usar a seguinte declaração:

```
FILE *Arquivo;
```

Pela declaração anterior, passa a existir uma variável de nome **Arquivo**, que é ponteiro para um arquivo.

Conforme citado, a função que abre um arquivo em C é a função **fopen()**, que devolve o valor **null** (nulo) ou um ponteiro associado ao arquivo, devendo ser passado para função o nome físico do arquivo e o modo como este arquivo deve ser aberto.

```
Arquivo = fopen ("texto.txt","w");
```

Com a instrução acima, está sendo aberto um arquivo de nome "texto.txt", no disco, habilitado apenas para escrita (w-write). Utilizando-se técnicas de verificação, pode-se codificar a instrução acima da seguinte maneira:

```
if ((Arquivo = fopen("texto.txt","w")) == NULL)
{
 puts ("Arquivo não pode ser aberto...");
 exit (1);
}
```

Além do modo de escrita, a linguagem C permite o uso de alguns valores padronizados para o modo de manipulação de arquivos, conforme mostra a **Tabela 15**.

Modo	Ação
r	*Abre um arquivo texto para leitura*
w	*Cria um texto para escrita*
a	*Anexa um arquivo texto*
rb	*Abre um arquivo binário para leitura*
wb	*Cria um arquivo binário para escrita*
r+	*Anexa um arquivo binário*
w+	*Cria texto para leitura/escrita*
a+	*Anexa texto para leitura/escrita*
rb+	*Abre um arquivo binário para leitura/escrita*
wb+	*Cria um arquivo binário para leitura/escrita*
ab+	*Anexa um arquivo binário para leitura/escrita*

Tabela 15 - Modos para manipulação de arquivos

Para o esvaziamento do buffer de um arquivo é utilizada a função **fclose()**, que associa-se diretamente ao nome lógico do arquivo (**STREAM**).

```
fclose (Arquivo);
```

```
// Programa que efetua leitura do teclado e grava em
// arquivo
#include <stdio.h>
#include <conio.h>
#include <stdlib.H>
void main (void)
{
 FILE *Arquivo;
 char Caractere, Nome[20];
 clrscr();
 printf("Nome do arquivo? ");
 gets(Nome);
 if ((Arquivo = fopen (Nome,"w")) == NULL)
 {
  printf ("Erro abertura\n");
  printf ("Arquivo : %s\n",Nome);
  exit (1);
 }
 do
 {
  Caractere = getche();
  if (Caractere == 13)
  {
   putc('\n',Arquivo);
   puts("");
  }
  else
      putc(Caractere,Arquivo);
  } while (Caractere != 26);
  fclose(Arquivo);
}
```

Programa 41 – Manipulação de arquivo-texto para escrita

```
// Programa que efetua leitura do arquivo e apresenta na
// tela

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

void main (void)
{
 FILE *Arquivo;
 char Nome[20];
 char Caractere;
 clrscr ();
 printf ("Nome do arquivo? ");
 gets (Nome);
```

```
if ((Arquivo = fopen(Nome,"r")) == NULL)
 {
  printf ("Erro de abertura\n");
  printf ("Arquivo: %s", Nome);
  exit (1);
 }
 Caractere = getc(Arquivo);
 while (Caractere != EOF)
 {
  putchar (Caractere);
  Caractere = getc(Arquivo);
 }
 fclose (Arquivo);
 getch ();
}
```

Programa 42 – Manipulação de arquivo-texto para leitura

Além da manipulação de arquivos do tipo texto, pode-se ler e escrever estruturas maiores que 1 `byte`, usando as funções **`fread()`** e **`fwrite()`**, conforme os protótipos seguintes.

`fread` (buffer, tamanhoembytes, quantidade, ponteirodearquivo)
`fwrite` (buffer, tamanhoembytes, quantidade, ponteirodearquivo)

O `buffer` é um endereço de memória da estrutura de onde deve ser lido ou onde devem ser escritos os valores (`fread()` e `fwrite()`, respectivamente). O `tamanhoembytes` é um valor numérico que define o número de `bytes` da estrutura que deve ser lida/escrita. A `quantidade` é o número de estruturas que devem ser lidas ou escritas em cada processo de `fread` ou `fwrite`. O `ponteirodearquivo` é o ponteiro do arquivo de onde deve ser lida ou escrita uma estrutura.

5 Gráficos

5.1 Biblioteca gráfica da linguagem C

Nada como um gráfico para facilitar o entendimento de coisas abstratas. Em Computação, assim como em Engenharia e outras áreas, faz-se uso de modelos gráficos como DFD, plantas e outros, para facilitar o planejamento de sistemas, construções e outros.

Cada compilador C tem sua biblioteca gráfica específica. Isto se deve ao fato da resolução gráfica ser uma característica dependente da máquina (*hardware*). De modo geral, é extremamente complexo o tratamento de todos os tipos de dispositivos gráficos em uma única biblioteca gráfica. Neste contexto, a biblioteca para as versões Borland Turbo C, é a **"graphics.h"**, que é muito semelhante à `UNIT` gráfica "`graph.tpu`" disponível no Turbo Pascal.

Certamente, poucos profissionais de computação, quando observam e/ou trabalham com sistemas gráficos baseados em janelas, tais como ***Microsoft Windows 95*** e ***OS2/Warp***, se preocupam em compreender os fundamentos gráficos que possibilitaram a construção dos mesmos. Neste contexto, torna-se interessante, e até mesmo fundamental, conhecer uma biblioteca gráfica básica e, a partir dos recursos por ela fornecidos, implementar objetos gráficos avançados que se comportem como os encontrados nos ambientes supra-citados. Assim, neste material didático é utilizada a biblioteca **graphics.h** do ambiente Borland Turbo C, de cuja as principais funções são apresentadas a seguir:

- **`initgraph`**: inicializa o modo gráfico e necessita de 3 parâmetros de configuração. O `'graphdriver'` indica o *driver* gráfico de cada monitor (EGA,VGA,SVGA e outros). Estes *drivers* estão disponíveis em arquivos com extensão **`.BGI`**. O parâmetro `'graphmode'` define o modo gráfico mais indicado para cada monitor. O parâmetro `'pathstring'` indica o caminho dos diretórios onde se encontra o *driver* gráfico;
- **`DETECT`**: macro que efetua a auto-detecção de modo gráfico, garantindo a portabilidade;
- **`graphresult`**: função que retorna um código de erro para a última operação gráfica executada num programa;
- **`setviewport`**: define uma região da tela gráfica como área ativa;
- **`settextstyle`**: define o estilo (tamanho, tipo, direção) dos textos (`strings`) que serão exibidos no modo gráfico;
- **`setcolor`**: especifica uma cor (paleta de cores) para trabalho no modo gráfico;
- **`setfillstyle`**: ajusta um padrão e cor para preenchimento de figuras;

- **line**: desenha uma linha entre dois pontos especificados. Desenha a linha de (X1, Y1) a (X2, Y2) usando a cor e estilo correntes;
- **putpixel**: acende um pixel (*picture element*) num ponto específico;
- **getpixel**: retorna a cor de um pixel (X,Y) da tela gráfica;
- **rectangle**: desenha um retângulo em modo gráfico utilizando a cor corrente;
- **lineto**: desenha uma linha entre a posição corrente do indicador gráfico e os pontos (X,Y) definidos utilizando a cor corrente;
- **moveto**: move o indicador gráfico para a posição (X,Y) definida;
- **bar**: desenha uma barra (retângulo preenchido) em modo gráfico utilizando a cor corrente;
- **circle**: traça um círculo, utilizando a cor corrente, de acordo com um raio especificado;
- **floodfill**: preenche uma figura fechada a partir de um ponto;
- **outtextxy**: escreve uma `string` numa certa posição da tela, em modo gráfico;
- **outtext**: escreve uma `string` na posição corrente do indicador de tela, em modo gráfico;
- **getmaxx**: retorna a resolução (número máximo de pixels) na horizontal (X), da tela gráfica;
- **getmaxy**: retorna a resolução (número máximo de pixels) na vertical (Y), da tela gráfica;
- **closegraph**: sai do modo gráfico.

```
// Apresenta o uso das funções gráficas
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <graphics.h>

void DefineJanela(int X1, int Y1, int X2, int Y2,
     unsigned char CorFundo, unsigned char CorBorda);

void main (void)
{
 int DriverGrafico, ModoGrafico, Erro, X,Y;
 int TrocaCor,ContadorPixels;
 unsigned char Cor;
 DriverGrafico = DETECT;
 initgraph (&DriverGrafico,&ModoGrafico,"c:\\btcpp30\\bgi");
 Erro = graphresult ();
 if (Erro != grOk)
 {
  printf ("Erro de Abertura do Modo Grafico"); exit (1);
 }
 // Definindo uma primeira janela (área gráfica) e usando
 // recursos cartesianos
 DefineJanela(0,0,300,220,7,15); setfillstyle (1,0);
 setcolor(0); circle(50,50,20);
```

```
setfillstyle (1,4); setcolor(4);
circle(100,50,20); floodfill(100,50,4);
setfillstyle(1,2); rectangle(200,30,280,70);
setfillstyle(1,14); setcolor(14);
line(30,140,30,200); line(30,200,130,200);
line(130,200,30,140);
setfillstyle(1,4); setcolor(4);
moveto(30,140); lineto(100,140);
setfillstyle(1,3); setcolor(3); bar(200,130,280,170);
getch();

// Definindo uma segunda janela (área gráfica ativa) e
// usando recursos de textos
DefineJanela(100,100,400,320,6,15);
setfillstyle(1,15); setcolor(15);
outtextxy(40,30,"Usando recursos graficos da");
outtextxy(40,42,"Linguagem C...");
setfillstyle(1,11); setcolor(11);
outtextxy(40,80,"Usando recursos graficos da");
outtextxy(40,92,"Linguagem C...");
setfillstyle(1,3); setcolor(3);
outtextxy(40,130,"Usando recursos graficos da");
outtextxy(40,142,"Linguagem C...");
setfillstyle(1,14); setcolor(14);
moveto(156,142); outtext("e C++");

settextstyle(0,HORIZ_DIR,3); outtextxy(30,170,"OK");
settextstyle(1,HORIZ_DIR,3); outtextxy(80,170,"OK");
settextstyle(2,HORIZ_DIR,3); outtextxy(130,170,"OK");
settextstyle(3,HORIZ_DIR,3); outtextxy(180,170,"OK");
settextstyle(4,HORIZ_DIR,3); outtextxy(230,170,"OK");
getch ();

// Definindo uma terceira janela (área gráfica) e usando
// recursos de pixels
DefineJanela(200,200,500,420,1,15);
randomize( ); ContadorPixels = 0; TrocaCor = 0;
do
{
 X = random(getmaxx( )); Y = random(getmaxy( ));
 if (!TrocaCor)
    Cor = 15;
 else
     Cor = 8;
 if ((getpixel(X,Y) != 8) && (getpixel(X,Y) != 15))
 {
  setfillstyle(1,Cor); setcolor(Cor); putpixel(X,Y,Cor);
 }
 if ((getpixel(X,Y) != 15) && (getpixel(X,Y) != 15))
 {
  setfillstyle(1,Cor); setcolor(Cor); putpixel(X,Y,Cor);
}
```

```
  ContadorPixels++;
  if (ContadorPixels == 100)
  {
   TrocaCor = !TrocaCor;
   ContadorPixels = 0;
  }
 } while (!kbhit( ));

 getch();
 closegraph ();
}

void DefineJanela(int X1, int Y1, int X2, int Y2,
                  unsigned char CorFundo,
                  unsigned char CorBorda)
{
 setviewport(0,0,getmaxx(),getmaxy(),1);
 setfillstyle(1,CorFundo); setcolor(CorFundo);
 bar(X1,Y1,X2,Y2);
 setfillstyle(1,CorBorda); setcolor(CorBorda);
 rectangle(X1,Y1,X2,Y2);
 setviewport(X1,Y1,X2,Y2,1);
}
```

Programa 43 – Demonstração de uso de recursos gráficos

5.2 Rotinas de mouse integradas ao modo gráfico

Mesmo com toda a capacidade de C, existem momentos em que é necessário acessar diretamente os recursos do sistema operacional e do ambiente hospedeiro. Freqüentemente, um dispositivo específico ou mesmo uma função do sistema operacional, precisam ser utilizados sem que haja uma forma direta de acesso com as rotinas existentes em alguma das bibliotecas da linguagem.

Cada processador, sistema operacional e ambiente hospedeiro têm seus próprios métodos de acesso aos recursos do sistema. Quando se deseja uma forma mais portável de manipulação destes recursos é aconselhado o uso de rotinas pré-definidas do sistema operacional, que se encontram disponíveis a partir de um vetor de interrupções. Para acessar essas rotinas deve-se carregar os registradores de uso geral com os parâmetros de chamada da rotina, e então executar a chamada da rotina via interrupção.

Em C existem funções para chamada de interrupção, tais como **`bdos()`**, **`int86()`** e **`intr()`**.

Para o programa exemplo apresentado a seguir, são acessadas as rotinas de manipulação de *mouse*, via interrupção **`0x33`**, usando a função **`intr`** pré-definida em C para acesso aos recursos do sistema operacional.

```
// Apresenta rotinas de mouse aliadas ao modo gráfico
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>
#include <graphics.h>

unsigned int IniciaMouse (void);
int CoordX (void);
int CoordY (void);
void IndicadorMouse (void);
void SemIndicadorMouse (void);
int BotaoCentral (void);
int BotaoDireita (void);
int BotaoEsquerda (void);

void main (void)
{
 int DriverGrafico, ModoGrafico, Erro;
 int X,Y;
 unsigned int MouseInstalado, Finaliza;
 int BotaoEsq, BotaoDir;

 DriverGrafico = DETECT;
 initgraph (&DriverGrafico,&ModoGrafico,"c:\\tcpp\\bgi");
 Erro = graphresult( );

 if (Erro != grOk)
 {
  printf ("Erro de Abertura do Modo Grafico");
  exit (1);
 }

 MouseInstalado = IniciaMouse( );
 if (!MouseInstalado)
    exit(1);

 setfillstyle(1,15); setcolor(15);
 rectangle(0,0,100,50);
 rectangle(0,60,100,110);
 rectangle(0,120,100,170);
 setfillstyle(1,11); setcolor(11);
 outtextxy(20,22,"Mensagem");
 outtextxy(20,82," Avisos ");
 outtextxy(20,142,"Finaliza");
 IndicadorMouse( );
 Finaliza = 0;
 do
 {
  BotaoEsq = 0;
  BotaoDir = 0;
  BotaoEsq = BotaoEsquerda( );
  BotaoDir = BotaoDireita( );
```

```
  if ((CoordX( ) >= 0) && (CoordX( ) <= 100) &&
      (CoordY( ) >= 0) && (CoordY( ) <= 50) &&
      (BotaoEsq != 0))
  {
   setfillstyle(1,7); setcolor(7);
   SemIndicadorMouse( );
   bar(200,80,460,100);
   setfillstyle(1,0); setcolor(0);
   outtextxy(215,88,"Bem vindo ao mundo gráfico...");
   IndicadorMouse( );
 }

 if ((CoordX( ) >= 0) && (CoordX( ) <= 100) &&
      (CoordY( ) >= 60) && (CoordY( ) <= 110) &&
      (BotaoEsq != 0))
  {
   setfillstyle(1,7); setcolor(7);
   SemIndicadorMouse( );
   bar(200,80,460,100);
   setfillstyle(1,4); setcolor(4);
   outtextxy(210,88,"Programe com cautela e atenção");
   IndicadorMouse( );
  }

  if ((CoordX() >= 0) && (CoordX() <= 100) &&
      (CoordY() >= 120) && (CoordY() <= 170) &&
      (BotaoEsq != 0))
     Finaliza = 1;
 } while ((!Finaliza) && (BotaoDir == 0));

 IndicadorMouse( );
 closegraph ( );
}



unsigned int IniciaMouse (void)
{
 struct REGPACK Registradores;
 Registradores.r_ax = 0;
 intr (0x33, &Registradores);
 return Registradores.r_ax;
}

int CoordX (void)
{
 struct REGPACK Registradores;
 Registradores.r_ax = 03;
 intr (0x33, &Registradores);
 return Registradores.r_cx;
}
```

```
int CoordY (void)
{
 struct REGPACK Registradores;
 Registradores.r_ax = 03;
 intr (0x33, &Registradores);
 return Registradores.r_dx;
}

void IndicadorMouse (void)
{
 union REGS Registradores;
 Registradores.x.ax = 0x01;  Registradores.x.bx = 2;
 int86(0x33,&Registradores,&Registradores);
}

void SemIndicadorMouse (void)
{
 union REGS Registradores;
 Registradores.x.ax = 0x02;
 int86 (0x33,&Registradores,&Registradores);
}

int BotaoCentral (void)
{
 union REGS Registradores;
 Registradores.x.ax = 0x05;
 Registradores.x.bx = 2;
 int86(0x33,&Registradores,&Registradores);
 return Registradores.x.bx;
}

int BotaoDireita (void)
{
 union REGS Registradores;
 Registradores.x.ax = 0x05;
 Registradores.x.bx = 1;
 int86(0x33,&Registradores,&Registradores);
 return Registradores.x.bx;
}

int BotaoEsquerda (void)
{
 union REGS Registradores;
 Registradores.x.ax = 0x05;
 Registradores.x.bx = 0;
 int86(0x33,&Registradores,&Registradores);
 return Registradores.x.bx;
}
```

Programa 44 – Demonstração de uso de recursos de mouse

O significado de cada rotina do programa é evidenciado pelo funcionamento das funções **`intr`** e **`int86`**. Tais funções executam rotinas de baixo nível do sistema operacional. Cada rotina diferente é especificada por um número. No exemplo, para executar os serviços da rotina de *mouse* utiliza-se a interrupção **`0x33`**. Para especificar qual serviço da rotina de interrupção deve ser executado, basta especificar um valor para o registrador **`AX`**.

Referências Bibliográficas

01. SCHILDT, HERBERT: *C Completo e Total*. São Paulo, McGrawHill, 1990.

02. SCHILDT, HERBERT: *Linguagem C: Guia do Usuário*. São Paulo, Editora McGrawHill, 1986.

03. AMMERAAL, LEENDERT: *Computação Gráfica para IBM-PC*. São Paulo, Editora Atlas, 1989.

04. SWAM, TOM: *Aprendendo C++.* São Paulo, McGrawHill, São Paulo, 1993.

05. HOLZNER, STEVEN: *Borland C++ Programação for Windows*. São Paulo, Makron Books, 1995.

06. PAPPAS, CHRIS H. & MURRAY, WILLIAM H.: *Borland C++*. São Paulo, Editora Makron Books, 1995.

07. TENENBAUM, AARON et al.: *Estruturas de Dados Usando C*. São Paulo, Editora Campus, 1989.

08. TENENBAUM, AARON et al.: *Data Structures Using C and C++*. New Jersey, USA, Prentice-Hall, 1996.

09. SCHILDT, HERBERT: *C e C++ Completo e Total*. São Paulo, Editora Makron Books, 1993.