Sumário

- Linguagem de Programação C

Breve Histórico

- A linguagem de programação C foi criada na década de 70, por Dennis Ritchie, que a implementou, pela primeira vez, num computador DEC PDP-11, rodando o sistema operacional UNIX.

Breve Histórico

- C é considerada uma linguagem de programação genérica e que pode ser utilizada para a criação de diversos tipos de programas, desde softwares de alto nível, como programas comerciais, até sistemas de baixo nível, como programas para automação industrial e sistemas para dispositivos embarcados.

Características Gerais

- C é uma linguagem de programação *case sensitive*.
- Os comandos em C são escritos em minúsculo.

Características Gerais

- Palavras-reservadas em C:

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

Características Gerais

- Uso de comentários:
  - Os comentários tornam o código-fonte de um programa mais legível, o que facilita as futuras manutenções.
  - Para se realizar um comentário em C, basta iniciar o texto que se quer comentar com uma barra e asterisco (/*) e terminar com um asterisco e uma barra (*/).

Características Gerais

- Exemplo de uso de comentários

```
1 /* Tudo o que estiver neste intervalo não será
  analisado pelo compilador.
2
3     int temperatura = 40;
4
5     Mesmo que seja uma declaração válida da
  linguagem. */
```

Estruturas

- Um registro é uma coleção de diversas variáveis, com tipos possivelmente diferentes.
- Em C, os registros são declarados através da palavra reservada **struct**.

Estruturas

- Sintaxe:

```
1 struct nome-da-estrutura {
2
3       tipo-da-variável variável;
4       tipo-da-variável variável;
5
6 };
```

Estruturas

- Exemplo de estrutura em C:

```
1 struct aluno {
2       char nome [30];
3       int matricula;
4       char curso [30];
5 };
```

Estruturas

- Para acessar um campo específico de uma estrutura, deve-se compor o nome do campo que se deseja acessar com o nome da estrutura.
- Essa composição é feita utilizando-se do ponto (.).

Estruturas

```
1   struct aluno a1, a2;
2   strcpy (a1.nome, "João");
3   a1.matricula = 123456;
4   strcpy (a1.curso, "Engenharia");
5   a2   =   a1;
6 if (a2 == a1) {
7     printf("São iguais.\n");
8 }
```

Estruturas 

- A função `strcpy` copia uma string de origem para uma string de destino, já que em C não é possível realizar esta operação diretamente. Por exemplo, a instrução a1.nome = “Joao” seria identificada como erro pelo compilador.
- Isto ocorre porque uma string é considerada como um vetor em C e, como tal, deve ser manipulada elemento a elemento.

Estruturas 

- A manipulação de strings em C é feita através das suas funções.
- A comparação de duas strings é possível através da função `strcmp (string1, string2)`, que compara a string1 com a string2.
- Se as duas forem idênticas, a função retorna zero. Se elas forem diferentes, a função retorna um outro valor.

Estruturas

- A Linguagem C ainda oferece um recurso de se renomear tipos de dados primitivos.
- Isto é feito através do comando `typedef`.
- Um exemplo de utilização para o `typedef` é:

```
typedef int TipoContador;
typedef char* TipoPalavra;
```

Estruturas

- É possível também usar o `typedef` nas estruturas. Exemplo:

```
1 struct tno{
2   char* palavra;
3   int contador;
4 };
5 typedef struct tno TipoNo;
```

Estruturas

- Variáveis podem então ser definidas utilizando-se esse novo nome, como uma espécie de apelido.
- No exemplo seguinte, a declaração de variável está sintaticamente correta:

```
TipoNo meuNo;

meuNo.contador = 1;
```

Vetores

- Os vetores, também chamados de **arrays**, são uma forma de armazenar vários dados, em uma única variável, sendo estes dados acessíveis por meio de um índice numérico.
- Os vetores devem **sempre** conter dados do mesmo tipo.
- A representação de matrizes segue a mesma regra dos vetores, diferenciando-se apenas com relação à quantidade de dimensões.

Vetores

- A seguir, um exemplo da declaração de vetores em C:

```
tipo-do-array nome-array [tamanho];
```

- Com este tipo de declaração, o compilador reserva um espaço na memória suficientemente grande para poder armazenar o número de elementos que foi especificado no tamanho.

Vetores

- Em C, o índice de um vetor começa em zero.
- Assim, num vetor de 100 posições, o índice assumirá valores no intervalo de 0 a 99.
- Tentar acessar um índice fora do intervalo, levará a um erro.

Ponteiros

- A manipulação de ponteiros em C é, ao mesmo tempo, considerada como uma das suas principais características e também motivo para afugentar novatos na linguagem.

Ponteiros

- Os ponteiros são utilizados comumente para resolver diversos problemas computacionais como, por exemplo, permitir que diferentes partes do código-fonte sejam facilmente compartilhadas e também para manipular diversas estruturas com alocação dinâmica, tais como listas ligadas e árvores.

Ponteiros

- Como regra geral, um ponteiro é uma variável que guarda o endereço de outras variáveis.
- Uma variável comum pode ser vista como uma caixa, em que é possível armazenar os valores de um determinado dado.
- Já um ponteiro pode ser visto como uma variável que guarda o endereço da caixa.

Ponteiros

- Em C, os ponteiros são identificados através de um asterisco (*).
- A sintaxe dos ponteiros é implementada da seguinte maneira:

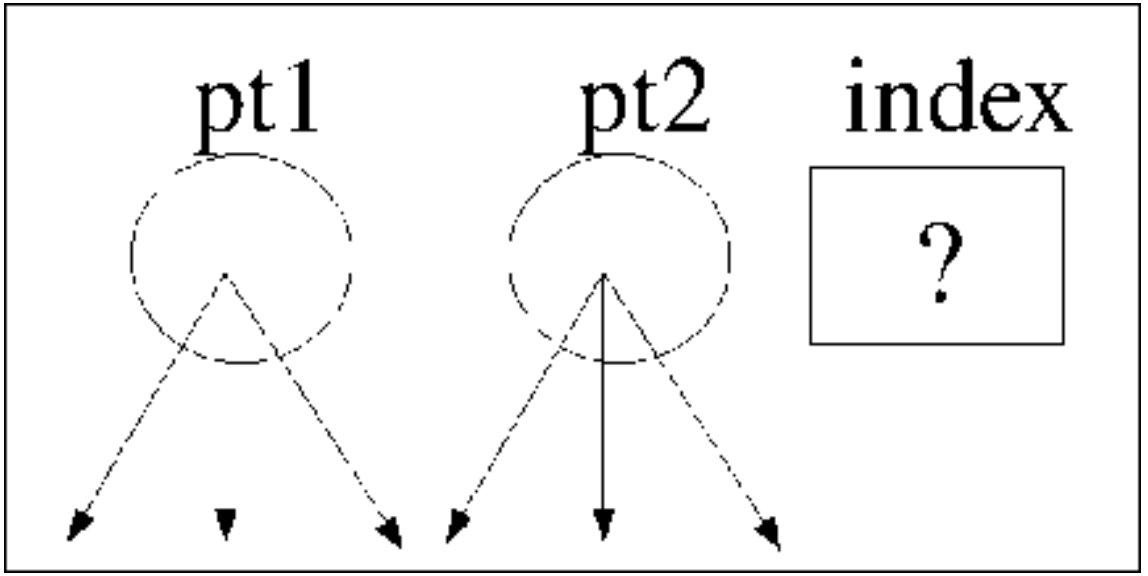
```
tipo-de-dado* NomeVar
```

Ponteiros

```
main ( ) {

int index, *pt1, *pt2;

index = 100;

pt1 = &index;

pt2 = pt1;

printf ("Os valores iniciais das variáveis index, pt1
  e pt2 são %d %d %d\n", index, *pt1, *pt2);

*pt1 = 50;

printf ("Os valores iniciais das variáveis index, pt1
  e pt2 são %d %d %d\n", index, *pt1, *pt2);

}
```

Ponteiros

- Exemplo de declaração de ponteiros:



Ponteiros

- No exemplo são declaradas três variáveis do tipo inteiro, sendo que as variáveis `pt1` e `pt2` são ponteiros para o tipo inteiro.
- A variável `index` foi criada e com espaço alocado, porém sem ter um valor, pois a mesma ainda não foi inicializada.
- Já os ponteiros `pt1` e `pt2` foram criados, mas ainda não estão apontando para lugar algum.

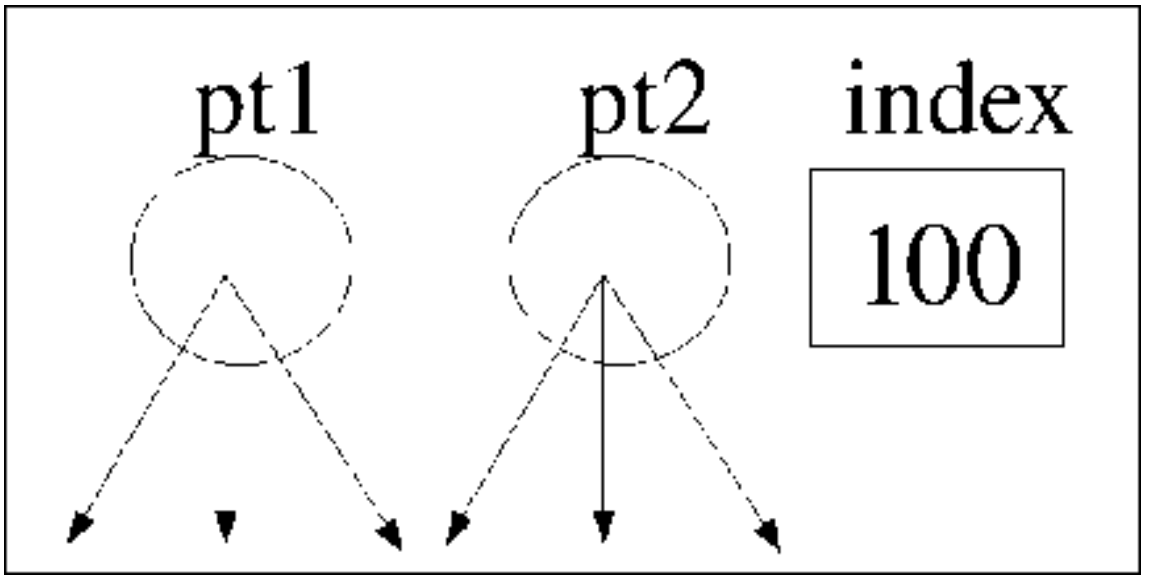
Ponteiros

- Os ponteiros `pt1` e `pt2` foram desenhados com uma elipse, diferentemente da variável `index` que foi desenhada com um retângulo, para indicar que aqueles não alocam na memória o mesmo espaço que uma variável comum aloca.
- Na verdade, os ponteiros armazenam apenas os endereços para os quais apontam.

Ponteiros

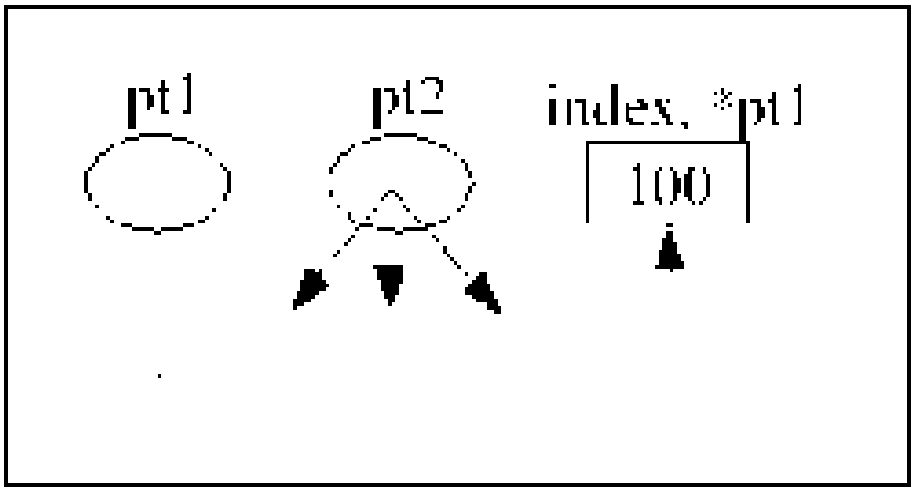
- A figura abaixo apresenta o estado das variáveis após a execução da linha:

```
index = 100;
```



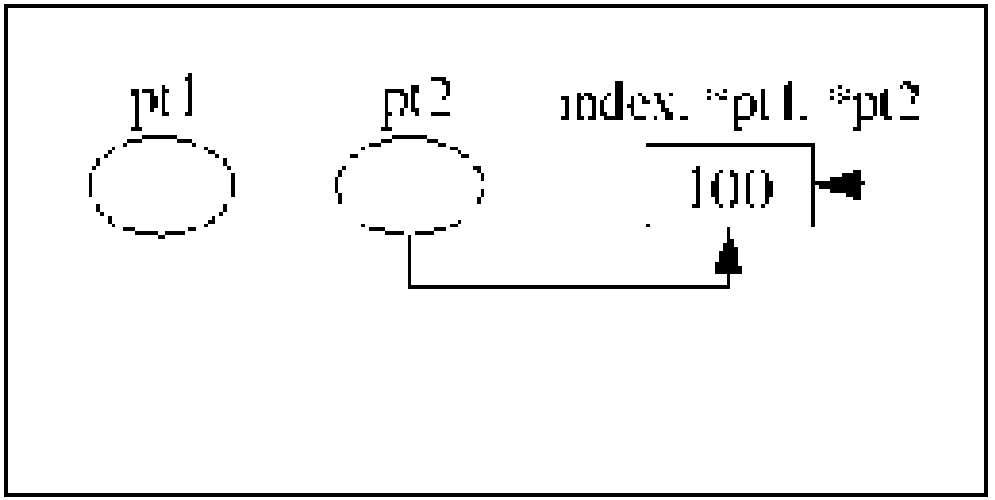
Ponteiros

- O ponteiro `pt1` recebe o endereço da variável `index`. Portanto, neste momento, o valor armazenado no ponteiro `pt1` é igual ao endereço da variável `index`.



Ponteiros

- No exemplo, o ponteiro `pt2` passa a apontar para o mesmo endereço do ponteiro `pt1`, ou seja, ambos os ponteiros passam a apontar para o endereço da variável `index`.



Ponteiros

- O comando abaixo imprime os conteúdos das três variáveis:

```
printf ("Os valores iniciais das variáveis
  index, pt1 e pt2 são %d %d %d\n", index,
  *pt1, *pt2);
```

- O endereço de memória apontado por `pt1` passa a ser igual a 50.

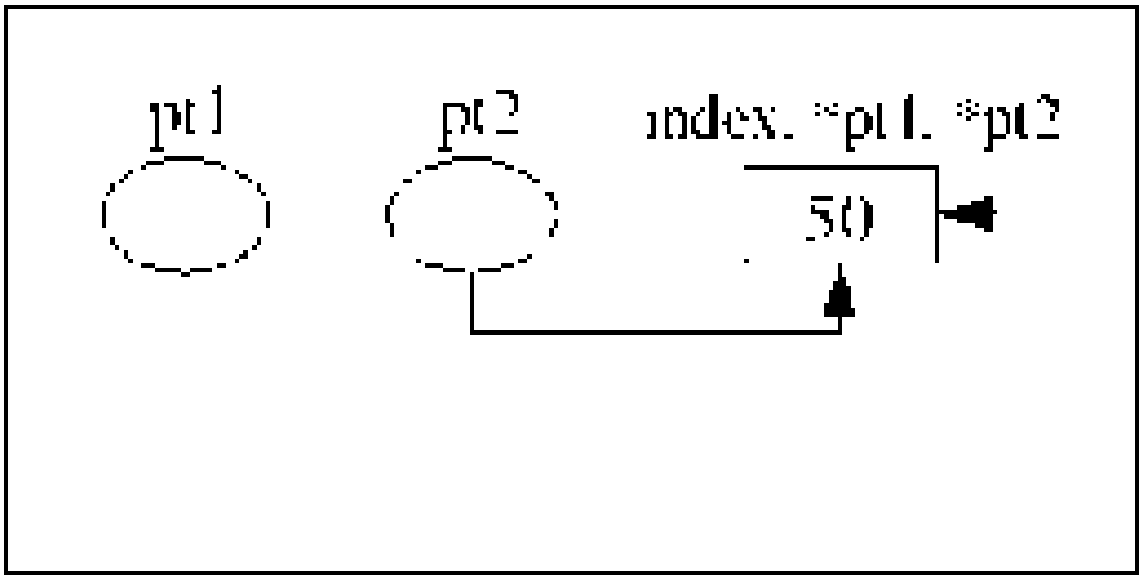
```
*pt1 = 50;
```

Ponteiros

- Portanto, como as três variáveis estão associadas ao mesmo endereço, as três variáveis passam a ter valor igual a 50.

Ponteiros

- Mudança automática das variáveis.



Alocação dinâmica

- Alocação dinâmica é a técnica de programação que permite alocar memória para variáveis quando o programa está sendo executado.
- Isto possibilita que o programador possa criar, por exemplo, um vetor ou uma matriz cujo tamanho será definido em tempo de execução, evitando, desta maneira, o desperdício de memória.

Alocação dinâmica

- A alocação dinâmica é bastante empregada em problemas que envolvem estruturas de dados como, por exemplo, pilhas, filas, árvores binárias e grafos.
- O padrão ANSI C define apenas 4 funções para o sistema de alocação dinâmica, disponíveis na biblioteca `stdlib.h`.
- As funções são: `malloc(.)`; `calloc(.)`; `realloc(.)` e `free(.)`.

Alocação dinâmica

- A função `malloc(.)` é empregada para alocar memória e possui a seguinte sintaxe:

```
void *malloc (unsigned int num);
```

Alocação dinâmica

- Aquela função recebe como parâmetro `num`, que é o número de *bytes* de memória que se deseja alocar.
- O tipo deste parâmetro é definido em `stdlib.h` como sendo um inteiro sem sinal. Vale ressaltar que esta função retorna um ponteiro que é do tipo `void`, podendo ser atribuído a qualquer tipo de ponteiro.

Alocação dinâmica

- A memória alocada por esta função é conseguida através do *heap*, que é a região de memória livre do computador.
- Desta maneira, é necessário verificar se a memória livre é suficiente para o armazenamento.

Alocação dinâmica

- Isto pode ser feito através do seguinte trecho de código:

```
if ((vet = (int *) malloc (tamanho)) == NULL)
  {
    printf ("Memória insuficiente");
    exit (1);
}
```

Alocação dinâmica

- Caso a função `malloc(.)` devolva o valor nulo significa que não há memória disponível e, neste caso, o programa irá imprimir na tela a mensagem “Memória insuficiente”.
- Para evitar que isso ocorra, ou seja, que haja indisponibilidade de memória, o programador é responsável por liberar **toda** a memória alocada de forma dinâmica. Isto é feito através da função `free(.)`.

Alocação dinâmica

- A função `exit(.)` pode ser chamada de qualquer lugar do programa e faz com que o programa termine e retorne, para o sistema operacional, o código de retorno.
- Por convenção, a função retorna zero no caso de um término normal do programa e retorna um número não nulo, no caso de ter ocorrido algum problema na execução do programa.

Alocação dinâmica

- É uma prática comum de programação sempre encerrar o programa quando uma chamada à função `malloc(.)` retornar um valor nulo.
- Isso pode ser feito usando-se a função `exit(.)`.

Alocação dinâmica

- A sintaxe da função `free(.)` é a seguinte:

```
void free (void *pt);
```

- Sendo que `pt` é um ponteiro que fora alocado previamente.
- No caso da alocação feita anteriormente, a função `free(.)` deve ser chamada da seguinte forma:

```
free(pt)
```

Funções

- Uma função fornece uma maneira conveniente para encapsular trechos de programas que podem ser utilizados sem que haja a preocupação sobre a sua implementação.
- Por exemplo, ao se utilizar a função `printf(.)` para imprimir algo na tela, o programador não sabe como a função foi implementada, mas apenas como utilizá-la.

Funções

- A linguagem C faz uso de funções de maneira fácil, conveniente e eficiente.
- Muitas vezes, as funções são utilizadas como forma de quebrar a complexidade dos problemas e aumentar a legibilidade, tornando o programa mais estruturado.

Funções

- Em C, a sintaxe de uma função é:

```
tipo-de-retorno nome-da-função
  (parâmetros, se houver) {
   corpo-da-função;
}
```

Funções

- As declarações de funções podem aparecer em qualquer ordem num arquivo, ou mesmo em arquivos separados.
- O tipo-de-retorno é o tipo da variável que a função vai retornar.
- Uma função que não tiver o tipo de retorno declarado, será considerada do tipo inteiro (`int`).

Funções

- A seguir, um exemplo de programa em C, que faz o cálculo de conversão de uma temperatura de Célsius para Fahrenheit.

Funções

```
#include <stdio.h>

float conversao
  (float tf) {
      float tc;
      tc = (tf - 32)
  * 5 / 9;
      return (tc);
  }
```

```
int main () {
    float valor;
    valor =
conversao (-40);
    printf ("O
valor é %f\n",
valor);
    return 0;
}
```

Tipo void

- Nem todas as funções precisam retornar algum valor. Em C, para indicar que uma função não terá retorno, basta indicar no início da função a palavra `void`.
- A seguir, é apresentada a sintaxe básica de uma declaração de função sem retorno:

```
void nome-da-função (parâmetros);
```

Arquivos

- As informações no computador são processadas e armazenadas temporariamente na memória principal.
- Essa memória é dita **volátil**, pois ao desligar o computador, todas as informações armazenadas nela são perdidas.

Arquivos

- Para que isto não ocorra, é necessário o armazenamento dos dados num dispositivo de memória secundária, não-volátil, como o disco rígido, utilizando-se de estruturas especiais chamadas de arquivos.
- A manipulação de arquivos envolve três etapas: criação ou abertura do arquivo; gravação ou leitura de dados no arquivo; e o fechamento do arquivo.

Arquivos

- Na etapa inicial é verificada a existência do arquivo.
- Se o arquivo já existe, ele será aberto para ser lido ou editado.
- Caso contrário, ele deverá ser criado, de acordo com o tipo escolhido.
- Os arquivos podem ser de dois tipos: texto ou binário.

Arquivos

- A escolha do tipo de arquivo será conforme a aplicação a ser desenvolvida.
- Se houver necessidade de que o conteúdo do arquivo seja lido por outras aplicações, deve-se escolher o tipo texto.

Arquivos

- Caso o armazenamento seja sobre dados que devam ser protegidos, o tipo mais indicado é o binário, pois além de ocupar menos espaço, ele também dificulta a leitura por outras pessoas, pois estas precisarão ter acesso ao programa que faz a interpretação do arquivo binário.

Arquivos

- Uma vez que o arquivo tenha sido aberto ou criado, inicia-se a segunda etapa da manipulação, que consiste em realizar operação de leitura e gravação de dados.
- A linguagem C tem um vasto conjunto de funções para manipular arquivos, sejam eles do tipo texto, ou do tipo binário.
- A biblioteca `stdio.h` possui as definições das funções de manipulação de arquivos.

Arquivos

- Em C, os arquivos não podem ser manipulados diretamente, mas através de uma variável do tipo FILE.
- A declaração deste tipo é feita da seguinte maneira:

```
FILE *fp;
```

Arquivos

- Uma vez que o ponteiro para manipular os arquivos tenha sido criado, deve-se tentar abrir ou criar o arquivo através da função `fopen(.)`.

A sintaxe da utilização da função `fopen(.)` é:

```
fp = fopen ("nome.extensão","modo
 +tipo");
```

Arquivos

- Opções para criação ou abertura de arquivos.

Modo	Significado
r	abre um arquivo-texto para leitura
w	cria um arquivo-texto para escrita
a	anexa a um arquivo-texto
rb	abre um arquivo binário para leitura
wb	cria um arquivo binário para escrita
ab	anexa um arquivo binário
r+	abre um arquivo-texto para leitura/escrita
w+	cria um arquivo-texto para leitura/escrita
a+	anexa ou cria um arquivo-texto para leitura/escrita
r+b	abre um arquivo-binário para leitura/escrita
w+b	cria um arquivo binário para leitura/escrita
a+b	anexa um arquivo binário para leitura/escrita

Arquivos

- Uma vez que o arquivo tenha sido aberto ou criado, é possível começar a gravar ou a ler os dados deste arquivo.
- Uma forma de realizar a leitura dos dados em um arquivo é através da função `fscanf(.)`.
- A seguir, mostra-se um exemplo de utilização da função `fscanf(.)`.

```
fscanf (fp, "%d", &texto);
```

Arquivos

- Para a gravação no arquivo, pode-se utilizar a função `fprintf(.)`, que é bastante parecida com função `printf(.)`.
- A diferença é que existe um parâmetro, na primeira posição, que indica qual é o arquivo para onde os dados serão gravados.

Arquivos

- A seguir, apresenta-se um exemplo de como se utilizar a função `fprintf(.)` para gravar no arquivo:

```
fprintf (fp, "O conteúdo de %d será
  gravado no arquivo.\n", variavel);
```

Arquivos

- O fechamento do arquivo é realizado através do comando `fclose(.)`.
- Esta função escreve qualquer dado que ainda permaneça no *buffer* de disco no arquivo e, então, fecha normalmente o arquivo.
- A seguir, um exemplo de utilização do comando `fclose(.)`:

```
fclose (fp);
```