Estruturas de Dados

2. Registros.

Um registro é uma variável especial que contém diversas outras variáveis normalmente de tipos diferentes, agrupadas em uma unidade. Em linguagem C, os registros são especificados pela palavra-chave *struct*.

As variáveis internas contidas pela struct são denominadas membros da struct.

Sintaxe:

```
struct <identificador>
{
   <listagem dos tipos e membros>;
};

struct <identificador> <variavel>;
```

Exemplo de declaração de uma struct

```
struct ficha_de_aluno
{
   char nome[50];
   char disciplina[30];
   float nota_prova1;
   float nota_prova2;
};

struct ficha_de_aluno aluno;
```

Neste exemplo criamos a *struct* **ficha_de_aluno**.

Depois de criar a *struct* precisamos criar a variável que vai utilizá-la.

Para tanto criamos a variável **aluno**, que será do tipo **ficha_de_aluno**.

```
struct ficha_de_aluno aluno;
```

Vamos ver agora um código completo, extraído de “http://linguagemc.com.br/struct-em-c/”:

```
#include <stdio.h>

int main(void)
{
   /*Criando a struct */
   struct ficha_de_aluno
```

```
    {
       char nome[50];
       char disciplina[30];
       float nota_prova1;
       float nota_prova2;
    };

    /*Criando a variável aluno que será do
    tipo struct ficha_de_aluno */
    struct ficha_de_aluno aluno;

    printf("\n-------- Cadastro de aluno ---------\n\n\n");

    printf("Nome do aluno ......: ");
    fflush(stdin);

    /* usaremos o comando fgets() para ler strings, no caso o nome
       do aluno e a disciplina
       fgets(variavel, tamanho da string, entrada)
       como estamos lendo do teclado a entrada é stdin
       (entrada padrão), porém em outro caso, a entrada tambem
       poderia ser um arquivo
    */

    fgets(aluno.nome, 40, stdin);

    printf("Disciplina ......: ");
    fflush(stdin);
    fgets(aluno.disciplina, 40, stdin);

    printf("Informe a 1a. nota ..: ");

    scanf("%f", &aluno.nota_prova1);

    printf("Informe a 2a. nota ..: ");
    scanf("%f", &aluno.nota_prova2);

    printf("\n\n ------- Lendo os dados da struct -------\n\n");
    printf("Nome ............: %s", aluno.nome);
    printf("Disciplina .....: %s", aluno.disciplina);
    printf("Nota da Prova 1 ...: %.2f\n" , aluno.nota_prova1);
    printf("Nota da Prova 2 ...: %.2f\n" , aluno.nota_prova2);

    return(0);
}
```

2.1 Vetores de Struct

Uma struct pode ser associada a vetores. Assim podemos criar um vetor de structs, da mesma forma que criamos um vetor de inteiros. Para isso, no momento da criação da variável, especificamos quantos elementos o vetor vai conter, como no exemplo abaixo, para armazenar os dados de uma classe com até 50 alunos.

Exemplo de declaração de uma struct

```
struct ficha_de_aluno
{
   char nome[50];
   char disciplina[30];
   float nota_prova1;
   float nota_prova2;
};

struct ficha_de_aluno aluno;

struct ficha_de_aluno turma[50];
```

Observe que o exemplo acima criou uma variável aluno, contendo uma estrutura do tipo ficha_de_aluno, e uma outra variável chamada *turma*, que é um vetor com 50 estruturas do tipo ficha_de_aluno.

Para fazer referência ao campo `nome` do primeiro aluno, usa-se a notação `turma[0].nome`.

Obviamente o índice do vetor pode ser uma variável. O exemplo abaixo ilustra um trecho de código para listar todos os elementos do vetor *turma*, supondo que *NumAlunos* contém o número de alunos cadastrados.

Exemplo:

```
int i=0;

for (i=0; i < NumAlunos; i++)

{

     printf("\nNome: %s – Disciplina: %s - P1: %d – P2: %d",

            turma[i].nome, turma[i].disciplina,

            turma[i].nota_prova1, turma[i].nota_prova2);

}
```

2.2 Estruturas aninhadas

É possível usar em C uma estrutura dentro de outra estrutura. Isso é relativamente comum, quando existem cadastros com um conjunto de campos agrupados. Por exemplo, em um cadastro de alunos, podemos precisar armazenar o endereço de cada aluno, que é composto por uma série de informações. O mesmo conjunto de informações de endereço pode ser usado em cadastros de professores, servidores e fornecedores, por exemplo.

Exemplo de definição de estuturas aninhadas. A estrutura aluno contém uma estrutura ender.

```
struct ender
{
     char rua[50];
     int numero;
     char complemento[15];
     char bairro[30];
     char cidade[30];
     char UF[4];
     char CEP[10];
};

struct aluno
{
     char nome[50];
     int matricula;
     char nascimento[12];
     char sexo;
     char telefone[20];
     struct ender endereco;
};

struct aluno cadastro[50];
```

3. Manipulação de arquivos em C.

Os arquivos permitem gravar os dados de um programa de forma permanente em mídia digital.

Vantagens de utilizar arquivos

- Armazenamento permanente de dados: as informações permanecem disponíveis mesmo que o programa que as gravou tenha sido encerrado, ou seja, podem ser consultadas a qualquer momento.
- Grande quantidade dados pode ser armazenada: A quantidade de dados que pode ser armazenada depende apenas da capacidade disponível da mídia de armazenamento. Normalmente a capacidade da mídia é muito maior do que a capacidade disponível na memória RAM.
- Acesso concorrente: Vários programas podem acessar um arquivo de forma concorrente.

A linguagem C trata os arquivos como uma sequência de bytes. Esta sequência pode ser manipulada de várias formas e para tanto, existem funções em C para criar, ler e escrever o conteúdo de arquivos independente de quais sejam os dados armazenados.

Há duas formas de trabalhar com arquivos em C: com buffers ou sem buffers.

A forma mais usada e mais eficiente em termos gerais é o uso de entrada e saída com buffers. Isso otimiza as operações porque os dados são armazenados em buffers do sistema operacional até que um bloco de dados seja completado, para então acionar uma operação física de acesso ao disco. Dessa forma, o número de operações físicas de acesso ao disco é reduzida e o desempenho do sistema aumenta. Dessa forma, uma gravação em disco pode demorar algum tempo após a chamada da função pelo programa. As funções mais usadas para isso são:

Já a entrada e saída sem buffers não utiliza esses buffers do sistema operacional, e a cada chamada a operação é executada imediatamente pelo hardware, o que provoca um maior número de acessos ao HD. Em contrapartida, ao sair do comando de gravação (write), tem-se a certeza que o dado está no HD, e não em algum buffer do sistema operacional aguardando por mais dados até ser efetivada a gravação em disco. As principais funções para entrada e saída sem buffer são open(), close(), read() e write().

Nesta disciplina trabalharemos com a entrada e saída com buffers, por ser mais usual, mais simples e apresentar melhor desempenho. As principais funções para entrada e saída com buffers são: fopen(), fclose(), fscanf(), fprintf(), fgets(), fputs(), fread(), fwrite(). As informações detalhadas sobre cada função, podem ser obtidas no linux usando o comando man <nome da função>. Exemplo: man fopen

3.1 Tipos de arquivos

Em C trabalhamos com dois tipos de arquivos:

1) Arquivo texto: Armazena caracteres que podem ser mostrados diretamente na tela ou modificados por um editor de texto.

Exemplos de arquivos texto: documentos de texto, código fonte C, páginas XHTML.

2) Arquivo binário é uma sequência de bits que obedece regras do programa que o gerou.

Exemplos: Executáveis, documentos do Word, arquivos compactados.

3.2 O ponteiro para arquivo

Em C, o arquivo é manipulado através de um ponteiro especial para o arquivo.

A função deste ponteiro é “apontar” a localização de um registro.

Sintaxe:
FILE < *ponteiro >

O tipo FILE está definido na biblioteca stdio.h.

Exemplo de declaração de um ponteiro para arquivo em C:

FILE *pont_arq;

Lembrando que FILE deve ser escrito em letras maiúsculas.

3.3 Operações com arquivos do tipo texto

3.3.1 Abertura e fechamento de arquivos.

Para trabalhar com um arquivo, a primeira operação necessária é abrir este arquivo.

Sintaxe de abertura de arquivo:
< ponteiro > = fopen(“nome do arquivo”, “tipo de abertura”);

A função fopen recebe como parâmetros o nome do arquivo a ser aberto e o tipo de abertura a ser realizado.

Depois de aberto, realizamos as operações necessárias e fechamos o arquivo.

Para fechar o arquivo usamos a função fclose.

Sintaxe de fechamento de arquivo

fclose (< ponteiro >);

Lembrando que o ponteiro deve ser a mesma variável ponteiro associada ao comando de abertura de arquivo.

Tipos de abertura de arquivos

r: Permissão de abertura somente para leitura. É necessário que o arquivo já esteja presente no disco.

w: Permissão de abertura para escrita (gravação). Este código cria o arquivo caso ele não exista, e caso o mesmo exista ele recria o arquivo novamente fazendo com que o conteúdo seja perdido. Portanto devemos tomar muito cuidado ao usar esse tipo de abertura.

a: Permissão para abrir um arquivo texto para escrita(gravação), permite acrescentar novos dados ao final do arquivo. Caso não exista, ele será criado.

Exemplo de código para a criação de um arquivo texto com nome “arquivo.txt”

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

int main(void)
{
        // criando a variável ponteiro para o arquivo
        FILE *pont_arq;

        //abrindo o arquivo
        pont_arq = fopen("arquivo.txt", "a");

        // fechando arquivo
        fclose(pont_arq);

        //mensagem para o usuário
        printf("O arquivo foi criado com sucesso!");

        system("pause");
        return(0);
}
```

Problemas na abertura de arquivos

Na prática, nem sempre é possível abrir um arquivo. Podem ocorrer algumas situações que impedem essa abertura, por exemplo:

- Você está tentando abrir um arquivo no modo de leitura, mas o arquivo não existe;
- Você não tem permissão para ler ou gravar no arquivo;
- O arquivo está bloqueado por estar sendo usado por outro programa.

Quando o arquivo não pode ser aberto a função fopen retorna o valor NULL.

É altamente recomendável criar um trecho de código a fim de verificar se a abertura ocorreu com sucesso ou não.

Exemplo:

```
if (pont_arq == NULL)
{
    printf("ERRO! O arquivo não foi aberto!\n");
}
else
```

```
{
     printf("O arquivo foi aberto com sucesso!");
}
```

3.3.2 Gravação de dados em um arquivo.

A função **fprintf** armazena dados em um arquivo. Seu funcionamento é muito semelhante ao printf, a diferença principal é a existência de um parâmetro para informar o arquivo onde os dados serão armazenados.

Sintaxe:

fprintf(nome_do_ponteiro_para_o_arquivo, “%s”,variavel_string)

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *pont_arq; // cria variável ponteiro para o arquivo
    char palavra[20]; // variável do tipo string

    //abrindo o arquivo com tipo de abertura w
    pont_arq = fopen("arquivo_palavra.txt", "w");

    //testando se o arquivo foi realmente criado
    if(pont_arq == NULL)
    {
        printf("Erro na abertura do arquivo!");
        return 1;
    }

    printf("Escreva uma palavra para testar gravacao de arquivo: ");
    scanf("%s", palavra);

    //usando fprintf para armazenar a string no arquivo
    fprintf(pont_arq, "%s", palavra);

    //usando fclose para fechar o arquivo
    fclose(pont_arq);

    printf("Dados gravados com sucesso!");

    return(0);
}
```

3.3.3. Leitura de dados de arquivos

Leitura caracter por caracter – Função getc()

Faz a leitura de um caracter no arquivo.

Sintaxe:

getc(ponteiro_do_arquivo);

Para realizar a leitura de um arquivo inteiro caracter por caracter podemos usar getc dentro de um laço de repetição.

Exemplo

```
do
{
    //faz a leitura do caracter no arquivo apontado por pont_arq
    c = getc(pont_arq);

    //exibe o caracter lido na tela
    printf("%c" , c);

} while (c != EOF);
```

A leitura será realizada até que o final do arquivo seja encontrado.

Leitura de strings – Função fgets()

É utilizada para leitura de strings em um arquivo. Realiza a leitura dos caracteres até o final da linha quando encontra o caracter \n. A leitura é efetuada de tal forma que a string lida é armazenada em um ponteiro do tipo char. A função pode ser finalizada quando encontrar o final do arquivo, neste caso retorna o endereço da string lida. Se ocorrer algum erro na leitura do arquivo, a função retorna NULL.

```
//Leitura de arquivo
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        FILE *pont_arq;
        char texto_str[20];

        //abrindo o arquivo_frase em modo "somente leitura"
        pont_arq = fopen("arquivo_palavra.txt", "r");

        //enquanto não for fim de arquivo o looping será executado
        //e será impresso o texto
        while(fgets(texto_str, 20, pont_arq) != NULL)
            printf("%s", texto_str);

        //fechando o arquivo
        fclose(pont_arq);

        getch();
        return(0);
}
```

3.4 Operações com arquivos do tipo binário

3.4.1 Abertura e fechamento de arquivos.

Para trabalhar com um arquivo, a primeira operação necessária é abrir este arquivo.

Sintaxe de abertura de arquivo:
< ponteiro > = fopen(“nome do arquivo”, “tipo de abertura”);

A função fopen recebe como parâmetros o nome do arquivo a ser aberto e o tipo de abertura a ser realizado.

Depois de aberto, realizamos as operações necessárias e fechamos o arquivo.

Para fechar o arquivo usamos a função fclose.

Sintaxe de fechamento de arquivo

fclose (< ponteiro >);

Lembrando que o ponteiro deve ser a mesma variável ponteiro associada ao comando de abertura de arquivo.

Tipos de abertura de arquivos

r: Permissão de abertura somente para leitura. É necessário que o arquivo já esteja presente no disco.

w: Permissão de abertura para escrita (gravação). Este código cria o arquivo caso ele não exista, e caso o mesmo exista ele recria o arquivo novamente fazendo com que o conteúdo seja perdido. Portanto devemos tomar muito cuidado ao usar esse tipo de abertura.

a: Permissão para abrir um arquivo texto para escrita(gravação), permite acrescentar novos dados ao final do arquivo. Caso não exista, ele será criado.

3.4.2 Gravação de dados em um arquivo.

A função fwrite() grava blocos de dados de tamanho fixo em um arquivo.

Sintaxe:

size_t fwrite(const void *<ap_dados>, size_t tam_registro, size_t num_registros,

FILE *<ap_arquivo>);

A função fwrite escreve num_registros itens de dados, cada um com *tam_registro* bytes, para o arquivo apontado por ap_arquivo, pegando os dados no endereço apontado por ap_dados.

Em caso de sucesso, a função retorna o número de itens gravados. Esse número é igual ao número de bytes gravados somente se tam_registro = 1.

Exemplo:

```
// Função que grava o conteúdo dos registros apontados por
//   cadastro, que contém registros de alunos
//   i contém o número de registros a serem gravados
//   retorna 0 se a gravação ocorreu com sucesso
//   retorna 1 em caso de erro
int salvaCadastro(struct aluno *cadastro, int i)
{
     FILE *arq;
     int j;

     arq = fopen("alunos.txt","w");
     if (arq == NULL)
     {
          fprintf(stderr,"\nErro ao abrir arquivo alunos.txt");
          return (1);
     }
     else
     {
        j=fwrite(&cadastro, sizeof(struct aluno),i,arq);
        fclose(arq);
        if (j == i)
           return (0);
        eles
           return (1);
     }
}
```

3.4.3 Leitura de dados de um arquivo.

A função fread() lê blocos de dados de tamanho fixo em um arquivo.

Sintaxe:

size_t fread(const void *<ap_dados>, size_t tam_registro, size_t num_registros,

FILE *<ap_arquivo>);

A função fread lê num_registros itens de dados, cada um com tam_registro bytes, do arquivo apontado por ap_arquivo, armazenando os dados no endereço apontado por ap_dados.

Em caso de sucesso, a função retorna o número de itens lidos. Esse número é igual ao número de bytes gravados somente se tam_registro = 1.

Exemplo:

```
// Função que lê registros do tipo aluno do arquivo “alunos.txt”
//   e armazena na variável global cadastro
//   retorna o número de registros lidos.
//   retorna -1 em caso de erro
int carregaCadastro()
{
     FILE *arq;
     int j, lido;

     arq = fopen("alunos.txt","r");
     if (arq == NULL)
     {
          fprintf(stderr,"\nErro ao abrir arquivo alunos.txt");
          return (-1);
     }
     else
     {
          j = 0;
          do {
               lido = fread(&cadastro[j], sizeof(struct aluno), 1,
                              arq);
               j++;
          } while (lido == 1);
          fclose(arq);
     }
     return (j-1);
}
```