FACULDADE DE ENGENHARIA DE SOROCABA

ENGENHARIA DA COMPUTAÇÃO

PROGRAMAÇÃO ESTRUTURADA II

Módulo 1 – Estruturas e Uniões

PROFª. ANDRÉA

1. ESTRUTURAS

1.1 – Introdução

Quando nos deparamos com um problema onde desejamos agrupar um conjunto de tipos de dados **não similares** sob um único nome, nosso primeiro impulso seria usar uma matriz. Porém, como matrizes requerem que todos os seus elementos sejam do **mesmo** tipo, provavelmente forçaríamos a resolução do problema selecionando uma matriz para cada tipo de dado, resultando em um programa ineficiente.

O problema de agrupar dados desiguais em C é resolvido pelo uso de estruturas

Uma estrutura é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, colocadas juntas sob um único nome. (Estruturas são chamadas "registros" em algumas linguagens). É uma outra maneira de representação de dados em C, que servirá para a elaboração de um arquivo de dados na memória acessado através de uma lista encadeada.

Um exemplo tradicional de uma estrutura é o registro de uma folha de pagamento, onde um funcionário é descrito por um conjunto de atributos tais como:

- nome (string)
- nº do seu departamento (int)
- salário (float) e assim por diante.

Provavelmente, haverá outros funcionários, e você vai querer que o seu programa os guarde formando uma matriz de estruturas.

O uso de uma matriz de várias dimensões não resolveria o problema, pois todos os elementos de uma matriz devem ser de um tipo único, portanto deveríamos usar várias matrizes:

- uma de caracteres para os nomes
- uma de inteiros para número do departamento
- uma float para os salários e assim por diante.

Esta não seria uma forma prática de manejar um grupo de características que gostaríamos que tivessem um único nome: **funcionário**.

Uma estrutura consiste de um certo número de itens de dados, chamados membros da estrutura, que não necessitam ser de mesmo tipo, agrupados juntos.

1.2 – Definindo um Tipo Estrutura

Primeiro, devemos definir o tipo da estrutura que queremos criar. Uma estrutura pode conter qualquer número de membros de diferentes tipos. O programa deve avisar o compilador de como é formada uma estrutura antes de seu uso.

sintaxe:

```
struct  nome_estrutura{
        membros_estrutura;
        };
```

onde:

nome_estrutura – rótulo ou etiqueta da estrutura, segue as mesmas regras de nomes de variáveis.
membros_estrutura – lista de variáveis, contém o tipo e o nome de cada variável da estrutura.

Ex.:

```
struct  facil  {
        int     num;
        char    c;
        };
```

Estas instruções definem um novo tipo de dado chamado **struct facil**. Cada variável deste tipo será composta por dois elementos:

- uma variável inteira chamada **num**
- e uma variável char chamada **c**.

Esta instrução não declara qualquer variável, e então não é reservado nenhum espaço de memória. Somente é mostrado ao compilador como é formado o tipo struct facil.

A palavra **struct** informa ao compilador que um tipo de dado está sendo declarado e o nome **facil** é chamado rótulo e nomeia a estrutura que está sendo definida.

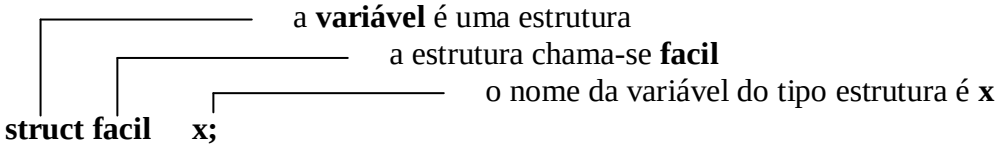
O rótulo não é o nome de uma variável, mas é o nome de um tipo. Os membros da estrutura devem estar entre chaves, e a instrução termina por ponto e vírgula.

UMA ESTRUTURA É UM TIPO DE DADO CUJO FORMATO É DEFINIDO PELO PROGRAMADOR.

1.3 - Declarando as Variáveis do Tipo Estrutura

Após definirmos nosso novo tipo de dado. Podemos, então, declarar uma ou mais variáveis deste tipo.

Ex.:



Em nosso exemplo, declaramos a variável **x** como sendo do tipo **struct facil**.

Esta instrução executa uma função similar às declarações de variáveis que já conhecemos como:

```
float   f;
int     num;
```

Ela solicita ao compilador a alocação de espaço de memória suficiente para armazenar a variável **x** que é do tipo **struct facil**, neste caso 6 bytes (4 bytes para o inteiro e 2 bytes para o caractere).

1.4 - Definindo e Declarando Estruturas

Se preferirmos, podemos definir e declarar a estrutura de uma única vez.
Ex.:

```
struct  facil  {
        int     num;
        char    c;
        }x;
```

Esta instrução define o tipo de estrutura **facil** e declara uma estrutura desse tipo, chamada **x**.

Dizemos que x é instância do tipo facil e contém 2 membros, um int chamado num e outro char chamado c

OBS.: C permite definir explicitamente **novos nomes** aos tipos de dados, utilizando a palavra-chave **typedef**. Declarações com typedef **não produzem novos tipos de dados**, apenas cria sinônimos.

sintaxe:

```
typedef     tipo_existente      novonome;
```

Ex.:
1)

```
typedef     float   real;   //o compilador reconhece real como outro nome para float
```

portanto, demos usar:

```
real    a, b;               // declaração de 2 variáveis do tipo real (float)
```

2)

```
struct  facil  {
        int     num;
        char    c;
        }x;

typedef   struct facil   facil;   //o compilador reconhece facil como outro nome p/ struct facil
facil   x;
```

ou

```
typedef struct facil {
        int     num;
        char    c;
        }facil;

facil   x;
```

ou

```
typedef struct {                    //sem etiqueta
        int     num;
        char    c;
        }facil;

facil   x;
```

1.5 - Acessando Membros da Estrutura

Agora que já criamos uma variável do tipo estrutura, precisamos acessar (referenciar) os seus membros.

Quando usamos uma matriz, podemos acessar um elemento individual através do índice: matriz[7].

Estruturas usam uma maneira de acesso diferente: o **operador ponto (.)**, que é também chamado **operador de associação**.

Ex.: A sintaxe apropriada para referenciar num que é parte da estrutura x, é:

x.num

O nome da variável que precede o ponto é o nome da estrutura e o nome que o segue é o de um membro específico da estrutura. As instruções:

```
x.num = 2;          // atribui 2 ao membro num da estrutura x
x.c = 'Z'           // atribui 'Z' ao membro c da estrutura x
```

O OPERADOR (.) CONECTA O NOME DA VARIÁVEL ESTRUTURA A UM MEMBRO DA ESTRUTURA.

Para exibir na tela:

```
printf("x.num = %i, x.c = %c \n", x.num, x.c);
```

A saída será:

```
x.num = 2, x.c = Z
```

1.6 - Múltiplas Estruturas de Mesmo Tipo

Do mesmo modo como podemos ter várias variáveis do tipo int em um programa, podemos também ter qualquer número de variáveis do tipo de uma estrutura predefinida.

Ex.: Vamos declarar duas variáveis, **x1** e **x2**, do tipo **struct facil**.

```
struct facil {
        int     num;
        char    c;
        };
struct facil    xl, x2;
```

ou, se preferir:

```
struct facil {
        int     num;
        char    c;
        }x1, x2;
```

O efeito é o mesmo, e o programa torna-se mais compacto.

1.7 - Definição de Estruturas Sem Rótulo ou Etiqueta

A convenção normal é a de usar a etiqueta da estrutura quando a expectativa é criar várias variáveis do mesmo tipo estrutura. Porém, se você espera usar uma única declaração de variável do tipo estrutura, você pode combinar a declaração com a definição da estrutura e omitir a etiqueta:

```
struct  {
        int     num;
        char    c;
        } x1, x2;
```

1.8 - Estruturas que Contém Matrizes

Podemos definir uma estrutura cujos membros incluam uma ou mais matrizes. A matriz pode ser de qualquer tipo de dados válido em C (char, int, float, double, etc.).

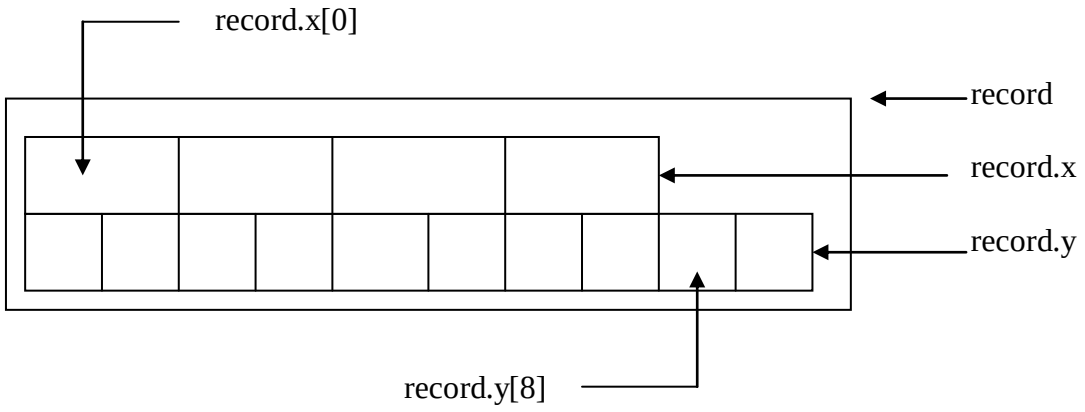
Ex.:

```
struct data{
        int     x[4];
        char    y[10];
        };
```

Definem uma estrutura do tipo **data** cujos membros são uma matriz inteira com 4 elementos chamada x e uma matriz de caracteres com 10 elementos chamada y. A seguir, poderíamos declarar uma estrutura chamada **record** baseada no tipo data:

```
struct data     record;
```

A organização desta estrutura é mostrada a seguir:



Observe que nessa figura os elementos da matriz x ocupam o dobro do espaço dos elementos da matriz y. Isto acontece porque, um elemento do tipo int normalmente exige 4 bytes, ao passo que um elemento do tipo char normalmente só exige 2 bytes.

O acesso aos elementos individuais de uma matriz que é um membro de uma estrutura é feito através de uma combinação de operadores de membros e subscritos de matrizes:

```
record.x[2] = 100;
record.y[1] = 'x';
```

O nome de uma matriz, quando é usado sem os colchetes, é um ponteiro para essa matriz. Como isto também se aplica às matrizes que são membros de estruturas, a expressão

```
record.y
```

é um ponteiro para o primeiro elemento da matriz y dentro da estrutura record. Podemos, portanto, imprimir o conteúdo de y [ ] na tela usando a instrução

```
puts(record.y);
```

1.9 – Exemplo - Criando uma Lista de Livros

A lista de livros compreende uma variedade de informações como:

- título
- autor
- editora
- número de páginas
- número do registro de biblioteca
- preço, etc.

Nosso objetivo é desenvolver um programa que controle um arquivo simples e mostre um dos mais úteis caminhos de organização de dados em C: uma matriz de estruturas.

Para simplificar o problema, a nossa lista será formada, a princípio, somente pelo título do livro, representado por uma matriz de caracteres e o número do registro de biblioteca representado por um inteiro. Estas informações são fornecidas pelo usuário através da entrada padrão (teclado) e depois impressas na saída padrão (vídeo).

Ex.: Na primeira versão a nossa lista será limitada em apenas dois livros.

```
#include <stdio.h>
#include <stdlib.h>                    // função atoi( )
struct  livro  {
     char       titulo[30];
     int        regnum;
};
main( )
{
struct livro     livro1,livro2;
char numstr[8];
printf("\n Livro  1  \n Digite titulo: ");
gets(livro1.titulo);
printf("Digite o numero do registro (3 digitos): ");
gets(numstr);
livro1.regnum  =  atoi(numstr);
printf("\n Livro  2  \n Digite titulo: ");
gets (1ivro2.titulo);
printf("Digite o numero do registro (3 digitos): ");
```

```
gets(numstr);
1ivro2.regnum  =  atoi (numstr);
printf("\n Lista de livros:\n");
printf("   Titulo: %s\n",livrol.titulo);
printf("   Numero do registro: %3i \n",livrol.regnum);
printf("   Titulo: %s\n",livro2.titulo);
printf("   Numero do registro: %3i \n",livro2.regnum);
}
```

Uma simples execução do programa terá a seguinte saída:

```
Livro  1.
Digite titulo: Helena
Digite o numero do registro (3 digitos): 102
Livro  2.
Digite titulo: Iracema
Digite o numero do registro (3 digitos): 321

Lista de livros:
Titulo: Helena
Numero do registro: 102
Titulo: Iracema
Numero do registro: 321
```

Poderíamos ter usado a função **scanf()** para ler o número do registro dos livros, como na instrução:

```
scanf("%i", &livro1.regnum);
```

que surtiria o mesmo efeito do uso de **gets()** e **atoi()**. Porém, preferimos usar gets() e atoi(), pois não provocam problemas com o buffer do teclado, se o usuário digitar espaços antes de digitar o número.

A função atoi( ), converte uma string ASCII num inteiro correspondente.

1.10 - Inicializando Estruturas

Nós já aprendemos como inicializar variáveis simples e matrizes:

```
int      num  =  5;
int      mat[ ]  =  {  1,  2,  3,  4,  5  };
```

Ex.: Uma versão modificada do programa, em que os dados dos 2 livros estão contidos na instrução de inicialização dentro do programa, em vez de serem solicitados ao usuário.

```
struct  livro  {
        char titulo[30];
        int regnum;
        };

main()
{
struct livro     livro1  =  {"Helena",102};
struct livro      livro2  =  {"Iracema",321};
```

```
printf("\nLista de livros:\n");
printf("Titulo: %s \n", livro1.titulo);
printf("Numero do registro: %3i \n", livrol.regnum);
printf("Titulo: %s \n", livro2.titulo);
printf("Numero do registro: %3i \n", livro2.regnum);
} // main
```

Aqui, depois da declaração usual do tipo estrutura, as duas variáveis estrutura são declaradas e inicializadas.

Da mesma forma como inicializamos matrizes, o sinal de igual é usado e em seguida a abertura da chave que irá conter a lista de valores. Os valores são separados por vírgulas.

1.11 - Atribuições entre Estruturas

Na versão original do C definida por Kernighan e Ritchie, é impossível atribuir o valor de uma variável estrutura a outra do mesmo tipo, usando uma simples expressão de atribuição.

Nas versões mais modernas de C, esta forma de atribuição já é possível. Isto é, se **livro1** e **livro2** são variáveis estrutura do mesmo tipo, a seguinte expressão pode ser usada:

livro2 = livro1;

O VALOR DE UMA VARIÁVEL ESTRUTURA PODE SER ATRIBUÍDO A OUTRA VARIÁVEL ESTRUTURA DO MESMO TIPO.

Quando executamos esta atribuição, os dados para os 2 livros serão exatamente os mesmos.

QUANDO ATRIBUÍMOS UMA ESTRUTURA A OUTRA, TODOS OS VALORES NA ESTRUTURA ESTÃO REALMENTE SENDO ATRIBUÍDOS PARA OS CORRESPONDENTES ELEMENTOS DA OUTRA ESTRUTURA.

Uma expressão de atribuição tão simples não pode ser usada para matrizes, que devem ser atribuídas elemento a elemento.

1.12 - Estruturas Aninhadas - Estruturas que contém Estruturas

Exatamente como podemos ter matrizes de matrizes, podemos ter estruturas que contêm outras estruturas. O que pode ser um poderoso caminho para a criação de tipos de dados complexos.

Como exemplo, imagine que os nossos livros são divididos em grupos, consistindo de um "dicionário" e um livro de "literatura".

Ex.: O programa cria uma estrutura de etiqueta **grupo**, que consiste de duas outras do tipo livro.

```
struct  livro  {
        char    titulo[30];
        int     regnum;
        };

struct  grupo  {
        struct livro     dicionario;
        struct livro     literatura;
        };

main()
{
struct grupo    grupo1 = {      {"Aurelio",134},
                                {"Iracema",321}       };
printf("\n Dicionario:\n");
printf("Titulo: %s \n", grupo1.dicionario.titulo);
printf("Nº do registro: %3i \n", grupo1.dicionario.regnum);
printf("\n Literatura:\n");
printf("Titulo: %s \n", grupo1.literátura.titulo);
printf("Nº do registro: %3i \n", grupo1.literatura.regnum);
}// main
```

Vamos analisar alguns detalhes deste programa:

- Primeiro, declaramos uma variável estrutura grupo1, do tipo grupo, e inicializamos esta estrutura com os valores mostrados. Quando uma matriz de várias dimensões é inicializada, usamos chaves dentro de chaves, do mesmo modo inicializamos estruturas dentro de estruturas.
- Segundo, note como acessamos um elemento da estrutura que é parte de outra estrutura. O operador ponto é usado duas vezes:

grupo1.dicionario.titulo

Isto referencia o elemento **titulo** da estrutura **dicionario** da estrutura **grupo1**.

Logicamente este processo não pára neste nível, podemos ter uma estrutura dentro de outra dentro de outra.

Tais construções aumentam o nome da variável, podendo descrever surpreendentemente o seu conteúdo.

1.13 - Matrizes de Estruturas

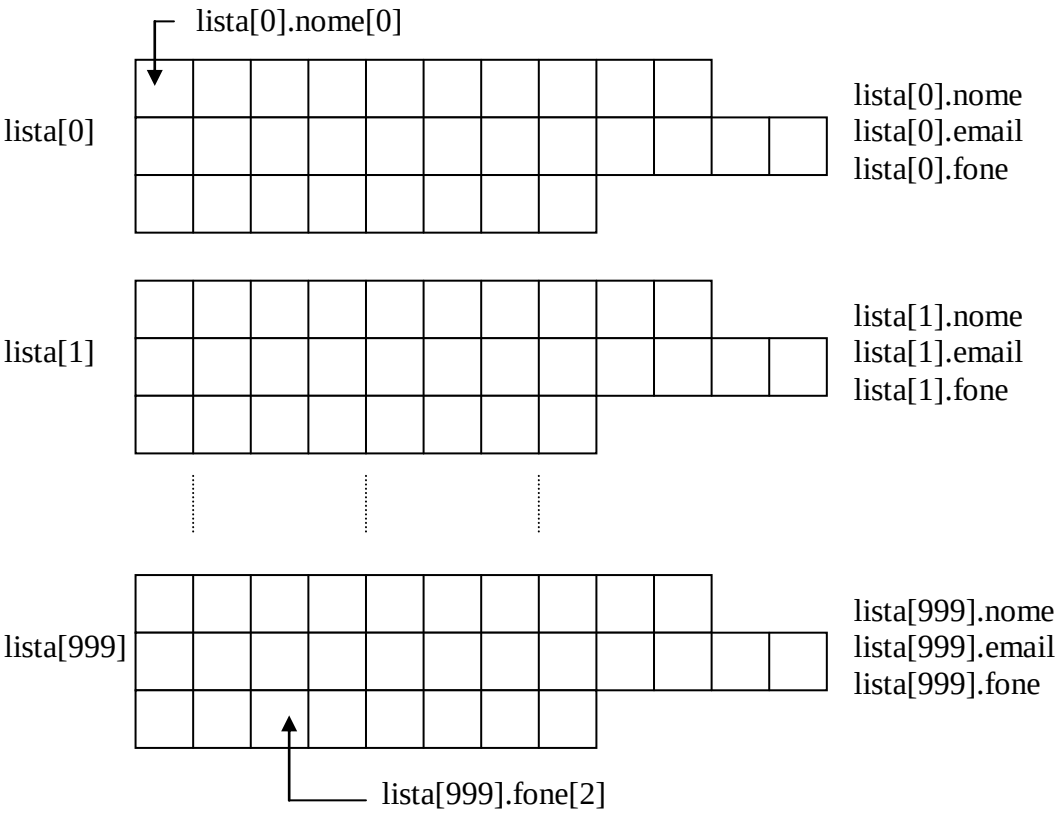
Por exemplo, em um programa que mantenha uma lista de emails e números de telefones, você poderia definir uma estrutura que armazenasse um nome, um email e um número:

```
struct entrada{
        char nome[10];
        char email[12];
        char fone[8];
        };
```

Porém, uma lista telefônica é composta de vários nomes, emails e números. Então, precisamos de uma matriz de estruturas. Depois que a estrutura estiver definida, podemos declarar uma matriz da seguinte maneira:

```
struct entrada      lista[1000];
```

Esta instrução declara uma matriz chamada lista que contém 1000 elementos. Cada elemento é uma estrutura do tipo entrada e é identificada por um índice, como em qualquer outra matriz. Cada uma dessas estruturas, por sua vez, contém três elementos, cada um dos quais é uma matriz do tipo char. Esse esquema relativamente complexo é ilustrado abaixo:



Da mesma forma que podemos atribuir o valor de uma variável estrutura a outra

```
lista[5]  =  lista[1]
```

Podemos transferir dados entre os elementos individuais das matrizes que são membros das estruturas:

```
lista[5].fone[1]  =  lista[2].fone[3];
```

Esta instrução move o segundo caractere do número de telefone armazenado em lista[5] para a quarta posição do número de telefone armazenado em lista[2]. (Não se esqueça que os índices começam em 0).

1.14 - Inicializando Estruturas Complexas

No caso de uma estrutura que contenha estruturas como membros, os valores de inicialização devem ser listados em ordem.

Ex.:

```
struct cliente {
        char firma[20];
        char contato[25];
        };

struct venda {
        struct cliente   comprador;
        char item[20];
        float quantia;
        } minhavenda   =   { { "Acme Industrias", "George Adams"},
                             "meias",
                             1000  };
```

Estas instruções realizam as seguintes inicializações:

- o membro da estrutura **minhavenda.comprador.firma** é inicializado com o string "Acme Industrias".
- o membro da estrutura **minhavenda.comprador.contato** é inicializado com o string "George Adams".
- o membro da estrutura **minhavenda.item** é inicializado com o string "meias".
- o membro da estrutura **minhavenda.quantia** é inicializado com o valor 1000.

No caso de matrizes de estruturas. Os dados de inicialização serão aplicados, pela ordem, às estruturas da matriz.

Ex. - declaramos uma matriz de estruturas do tipo venda e inicializamos os dois primeiros elementos da matriz (ou seja, as duas primeiras estruturas):

```
struct cliente {
        char firma[20];
        char contato[25];
        };

struct venda {
        struct cliente   comprador;
        char item[20];
        float quantia;
        };

struct venda    A1990[100] = {
        { { "Acme Indústrias", "George Adams" },
            "meias",
             1000       }
        { { "Wilson & Cia", "Edi Wilson" },
             "brincos",
             290        }        };
```

- o membro da estrutura **A1990[0].comprador.firma** será inicializado com o string "Acme Industrias".
- o membro da estrutura **A1990[0].comprador.contato** será inicializado com o string "George Adams".
- o membro da estrutura **A1990[0].item** será inicializado com o string "meias".
- o membro da estrutura **A1990[0].quantia** será inicializado com valor 1000.

- o membro da estrutura **A1990[1].comprador.firma** será inicializado com o string "Wilson & Cia".
- o membro da estrutura **A1990[1].comprador.contato** será inicializado com o string "Edi Wilson".
- o membro da estrutura **A1990[1].item** será inicializado com o string "brincos".
- o membro da estrutura **A1990[1].quantia** será inicializado com valor 290.

1.15 – Exercícios Propostos

1. Fazer um programa para **Sistema de Conta Bancária** – este programa se destina a controlar as contas de 10 clientes.

[1] Cadastro - receber os valores digitados pelo usuário. Apenas um registro é cadastrado por vez.
[2] Depósito - o acesso deve ser feito através do nº de conta corrente. Buscar o registro, mostrar o nome do cliente e o saldo para simples conferência, pedir o valor do depósito, fazer as alterações e apresentar na tela o saldo atualizado.
[3] Retirada - o acesso deve ser feito através do nº de conta corrente. Buscar o registro, mostrar o nome do cliente e o saldo para simples conferência, pedir o valor da retirada, fazer as alterações se possível (a retirada só será permitida, se houver saldo suficiente) e apresentar na tela o saldo atualizado.

```
struct cliente{
        char nome[30];
        int conta;
        float saldo;
};
```

2. Fazer um programa para **Diário Eletrônico** – este programa se destina a controlar as notas e a média de 10 alunos.

[1] Cadastro - receber os valores digitados pelo usuário, inicialmente notas e média=0. Apenas um registro é cadastrado por vez.
[2] Controle de Notas - o acesso deve ser feito através do RA. Buscar o registro, mostrar o nome do aluno para simples conferência, fazer as alterações das notas, calcular a média e apresentar na tela as notas e a média.

```
struct aluno{
        char nome[80];
        char RA[79];
        float nota[2];          //notas de provas – considerar 2 provas
        float media;            //média aritmética das provas
};
```

2. ESTRUTURAS COMPLEXAS

2.1 - Estruturas e Ponteiros

Podemos declarar ponteiros que apontem para estruturas e também usar ponteiros como membros de estruturas, ou ambos.

2.2 - Ponteiros para Estruturas

Os ponteiros para estruturas geralmente são usados para passar uma estrutura como argumento para uma função. Além disso, os ponteiros para estruturas também são usados em um método muito poderoso de armazenagem de dados conhecido como listas encadeadas.

Como um programa pode criar e usar ponteiros para estruturas? Em primeiro lugar, devemos definir uma estrutura.

Ex.:

```
struct peça {
        int     numero;
        char    nome[10];
};
```

A seguir, devemos declarar um ponteiro para o tipo peca.

```
struct peça     *p_peça;
```

O operador de indireção (*) na declaração significa que *p_peça* é um ponteiro para o tipo *peça*, não uma instância da estrutura do tipo peça.

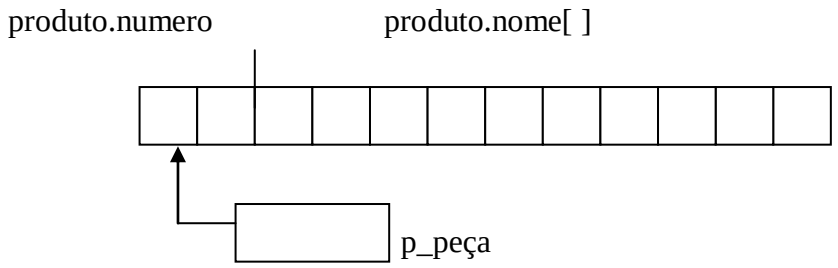
O ponteiro já pode ser inicializado? Não, porque a estrutura peça foi definida, mas nenhuma instância dessa estrutura foi declarada ainda. Lembre-se que é a declaração, não a definição, que reserva espaço na memória para a armazenagem de um objeto de dados. Como o ponteiro precisa de um endereço na memória para o qual possa apontar, teremos que declarar uma instância do tipo peça antes que algo possa apontar para ela.
Portanto, esta é a declaração:

```
struct peça     produto;
```

Agora já podemos inicializar o ponteiro:

```
p_peça  =  &produto;
```

Essa instrução atribui o endereço de produto a p_peça. A relação entre uma estrutura e um ponteiro para essa estrutura é mostrada na figura abaixo:



Um ponteiro para uma estrutura aponta para o primeiro byte dessa estrutura.

Agora que já temos um ponteiro apontando para a estrutura produto, podemos acessar os membros de uma estrutura usando um ponteiro de 3 maneiras:

2.2.1 - Operador de Acesso Indireto

É o mais comum. Que consiste dos símbolos -> (um sinal de menos seguido do símbolo de "maior que" - quando estes dois caracteres são usados juntos desta maneira, a linguagem C os interpreta como um único operador, não dois). O símbolo é colocado entre o nome do ponteiro e o nome do membro da estrutura.

Ex.:

p_peça->numero /* acessa o membro número da estrutura produto */

2.2.2 - Operador de Indireção (*)

Ex.:

p_peça é um ponteiro para a estrutura peça; portanto,
*p_peça refere-se a produto

Devemos aplicar o operador ponto (.) para acessarmos elementos individuais de produto.

(*p_peça).numero = 100; /* atribui o valor 100 a produto.numero */

O item *p_peça* deve estar entre parênteses porque o operador (.) tem maior precedência que o operador (*).

2.2.3 - Nome da Estrutura

Ex.:

produto.numero

Portanto, sendo p_peça um ponteiro para a estrutura peça, todas estas expressões são equivalentes:

produto.numero
(*p_peça).numero
p_peça->numero

2.3 - Ponteiros e Matrizes de Estruturas

Tanto as matrizes de estruturas como os ponteiros para estruturas são ferramentas de programação extremamente poderosas. Podemos também combinar esses dois métodos usando ponteiros para acessar estruturas que são elementos de matrizes.

Para ilustrar isso, vamos usar esta definição de estrutura de um exemplo anterior:

```
struct peça {
        int     numero;
        char    nome[10];
        };
```

Depois que a estrutura foi definida, podemos definir uma matriz do tipo peça, chamada dados:

```
struct peça     dados[100];
```

A seguir, podemos declarar um ponteiro para o tipo peça e inicializá-lo para apontar para a primeira estrutura contida na matriz:

```
struct peça     *p_peça;
p_peça  =  &dados[0];
```

Lembrando que o nome de uma matriz sem os colchetes é um ponteiro para o primeiro elemento dessa matriz. Portanto, a segunda linha também poderia ter sido escrita da seguinte forma:

```
p_peça  =  dados;
```

Agora já temos uma matriz de estruturas do tipo peça e um ponteiro para o primeiro elemento dessa matriz (ou seja, para a primeira estrutura da matriz). Poderíamos, por exemplo, imprimir o conteúdo desse primeiro elemento usando a instrução:

```
printf("%i      %s", p_peça->numero, p_peça->nome);
```

E se quiséssemos imprimir todos os elementos da matriz? Nesse caso, provavelmente usaríamos um loop for. Para acessar os membros usando notação de ponteiros, teríamos que alterar o ponteiro p_peça para que, a cada iteração do loop, ele apontasse para o próximo elemento da matriz (ou seja, para a próxima estrutura contida na matriz).

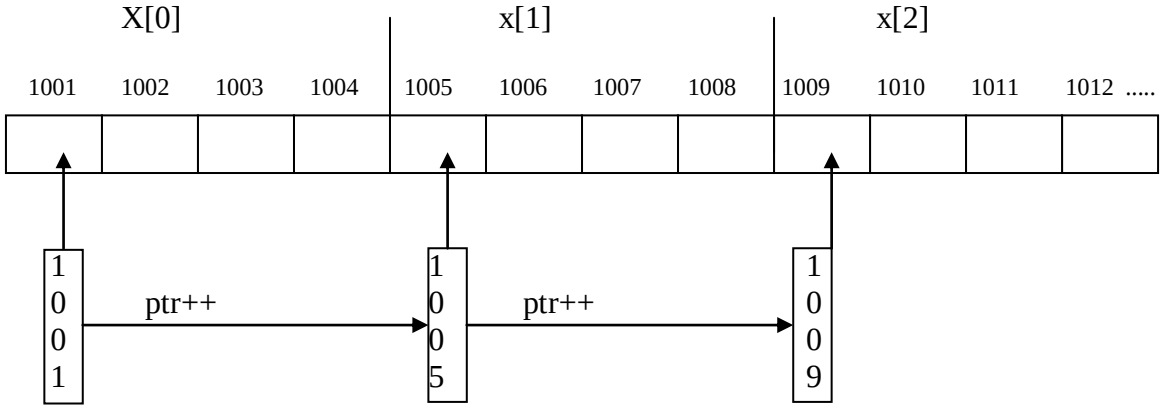
Para isso, a aritmética de ponteiros da linguagem C pode lhe ser útil. O operador unário de incremento (++) tem um significado especial quando é aplicado a um ponteiro, passando a significar "incrementar o ponteiro com um valor equivalente ao tamanho do objeto para o qual ele está apontando". Em outras palavras, se você tem um ponteiro ptr que aponta para um objeto de dados do tipo obj, a instrução:

```
ptr++;
```

tem o mesmo efeito de:

```
ptr += sizeof (obj);
```

Os elementos de uma matriz sempre são armazenados seqüencialmente na memória. Se um ponteiro estiver apontando para o elemento *n* de uma matriz, o uso do operador (++) para incrementá-lo fará com que ele passe a apontar para o elemento *n+1*. Isto é ilustrado na figura abaixo, que mostra uma matriz chamada x[ ] que consiste de elementos de quatro bytes (cada elemento pode ser, por exemplo, uma estrutura contendo dois membros do tipo char, cada um dos quais ocupa dois bytes). O ponteiro **ptr** foi inicializado para apontar para x[0] ; cada vez que é incrementado, **ptr** passa a apontar para o próximo elemento da matriz.



Isto significa que o seu programa pode avançar através de uma matriz de estruturas (ou, para ser exato, através de uma matriz de qualquer tipo) simplesmente incrementando um ponteiro. Este tipo de notação geralmente é mais fácil de usar e mais conciso do que o uso de subscritos de matrizes para realizar a mesma tarefa.

Ex. - acessa elementos sucessivos de uma matriz através do incremento de um ponteiro.

```
#include <stdio.h>
#define     MAX         4

struct peça {
        int numero;
        char nome[10];
        };
main()
{
struct peça     *p_peça,        dados[MAX] = {1, "camisa",
                                              2, "calça",
                                              3, "gravata",
                                              4, "casaco"     };
```

```
int       contagem;
p_peça  =  dados;                    // inicializa o ponteiro para o primeiro elemento da matriz
for (contagem = 0; contagem < MAX; contagem++)
   {
        printf("\n No endereço  %u : %i       %s", p_peça, p_peca->numero, p_peça->nome ) ;
        p_peça++;
   }
}// main
```

Teremos como saída, por exemplo:

```
No endereço   96: 1    camisa
No endereço 120: 2    calça
No endereço 144: 3    gravata
No endereço 168: 4    casaco
```

Inicialmente, o programa declara e inicializa uma matriz de estruturas chamada dados. A seguir, é definido um ponteiro chamado p_peça para apontar para uma estrutura do tipo peças. A primeira tarefa da função main( ) é inicializar o ponteiro p_peça para que ele aponte para a matriz dados declarada anteriormente. A seguir, todos os elementos dessa matriz são impressos na tela usando um loop for que incrementa o ponteiro a cada interação. O programa exibe também o endereço de cada elemento da matriz.

Observando os endereços mostrados, percebemos que todos os incrementos serão feitos em quantidades iguais, correspondendo ao tamanho da estrutura peças (na maioria dos sistemas, esse incremento será de 24 bytes). Isto ilustra claramente o fato de que o incremento de um ponteiro é feito com um valor igual ao tamanho do objeto de dados para o qual ele está apontando.

2.4 - Passando Estruturas como Argumentos para Funções

Como qualquer outro tipo de dados, uma estrutura pode ser passada como um argumento para uma função. O programa abaixo mostra como isso é feito.

Ex.:

```
#include <stdio.h>

struct dados{
        float    quantia;
        char     nome[80];
        } ;

void     print_reg (struct dados          x);

main( )
{
struct dados     reg;
printf("Digite o nome e o sobrenome do doador: ");
scanf("%s       %s", reg.nome, reg..snome);
printf("\n Digite a quantia doada: ");
scanf("%f", &reg.quantia);
print_reg( reg );
}// main
```

```
void    print_reg(struct dados          x)
{
printf("\O doador %s   doou R$%.2f.", x.nome, x.quantia);
}
```

Teremos como saída na tela:

```
Digite o nome e o sobrenome do doador: Carlos Silva
Digite a quantia doada: 1000.00
O doador Carlos Silva doou R$1000.00.
```

O protótipo da função deverá receber a estrutura, portanto devemos incluir os parâmetros adequados. No caso, é uma estrutura do tipo dados. As mesmas informações são repetidas no cabeçalho da função. Ao chamarmos a função, temos que informar o nome da instância dessa estrutura - no caso, **reg**. Isso é tudo. Passar uma estrutura para uma função não é muito diferente de passar uma variável simples. Alternativamente, podemos passar uma estrutura para uma função informando o seu endereço (ou seja, passando um ponteiro que aponte para a estrutura).

De fato, esta era a única maneira de usarmos uma estrutura como argumento nas versões mais antigas da linguagem C. Isso já não é necessário, mas é possível encontrarmos programas mais antigos que ainda usam esse método. Ao passarmos um ponteiro como argumento para uma função, teremos que usar o operador de acesso indireto (->) para acessarmos membros da estrutura de dentro da função.

Ex. – Novamente, usaremos o programa de livros, onde uma função obterá informações dos livros pelo usuário e outra irá imprimi-las.

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

typedef struct 1ivro {
        char titulo[30];
        int regnum;
        }livro;

livro   novonome();                     // função do tipo struct livro chamada novonome
void    listar (livro     liv);         // função void, cujo parâmetro é do tipo struct livro

main( )
{
livro   livro1, livro2;                 // variável estrutura chamada livro1, livro2

livro1 = novonome();                    // livro1 irá receber o retorno da função novonome
livro2 = novonome();                    // livro2 irá receber o retorno da função novonome
listar(livro1);
listar(livro2);
} // main
```

```
livro    novonome()
{
char numstr[8];
livro    livr;                                    // variável estrutura chamada livr

printf("\n Novo livro \n Digite titulo: ");
gets(livr.titulo);
printf("Digite o numero do registro (3 dígitos): ”);
gets(numstr);
livr.regnum  =  atoi(numstr);
return(livr);
}// novonome

void listar (livro         liv)
{
printf("\n Livro: \n");
printf("Titulo: %s ", liv.titulo);
printf("\n Nº do registro: %3i", liv.regnum);
} // listar
```

Visto que as duas funções, como também o programa main(), devem conhecer a estrutura livro, ela deve ser definida antes da main().

As funções main(), novonome() e listar() declaram internamente suas próprias variáveis estrutura, chamadas livro1 e livro2, livr e liv.

A função listar() recebe uma cópia da estrutura e a coloca num endereço conhecido somente por ela; **não** é a mesma estrutura declarada em main().

A função novonome() é chamada pelo programa principal para obter informações do usuário sobre os 2 livros. Esta função guarda as informações em uma variável interna, livr, e retorna o valor desta variável para o programa principal usando o comando **return**, exatamente como faria para devolver uma simples variável. A função novonome( ) deve ser declarada como sendo do tipo struct livro, visto que ela retorna um valor deste tipo.

O programa principal atribui o valor retornado por novonome() à variável estrutura livro1 e livro2. Finalmente main() chama a função listar() para imprimir os valores de livro1 e livro2, passando os valores destas duas estruturas para a função como variáveis.

A função listar() atribui estes valores à variável estrutura interna liv e acessa os elementos individuais desta estrutura para imprimir seus valores.

2.5 – Estruturas e Alocação Dinâmica

Para alocarmos memória dinamicamente na **main**, devemos declarar um ponteiro e utilizar malloc(), calloc() ou realloc(). Porém, quando utilizamos **função**, devemos lembrar que a memória não pode ser “conhecida” apenas na função, mas também na main. Portanto, devemos declarar um ponteiro na main e a função pode ser feita de 2 formas:

- **Chamada por Referência** – devemos passar o endereço do ponteiro declarado na main, portanto, a função receberá como parâmetro ponteiro para ponteiro.
- **Chamada por Valor** – só conseguimos trabalhar dessa forma, se a função **retornar o endereço** alocado para o ponteiro declarado na main, permitindo assim, o acesso deste à memória alocada. Caso contrário o ponteiro na main continuará NULL e ao sairmos da função perdemos a referência da memória alocada. Lembre-se que as variáveis (inclusive ponteiro) declaradas dentro da função, deixam de existir assim que a função termina.

Versão para Alocação feita na main( )

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

typedef struct 1ivro {
        char titulo[30];
        int regnum;
        }livro;

main( )
{
livro    *ptr=NULL;

if((ptr = (livro *) realloc(ptr, 10 * sizeof(livro))) == NULL)          // aloca 10 elementos
   {
    printf("Erro na alocação);
    exit(1);
   }
}//main
```

Versão utilizando Chamada por Referência

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

typedef struct 1ivro {
        char titulo[30];
        int regnum;
        }livro;

void aloca(livro **p, int tam);                    //passa o endereço do ponteiro declarado na main

main( )
{
livro    *ptr=NULL;

aloca(&ptr, 10);                                   //chamada por referencia
}//main
```

```
void aloca(livro **p, int tam)
{
if((*p=(livro*)realloc(*p, tam*sizeof(livro)))== NULL)
  {
   printf("Erro de alocacao");
   exit(1);
  }
}//aloca
```

Versão utilizando Chamada por Valor – com RETORNO do endereço alocado

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

typedef struct 1ivro {
        char titulo[30];
        int regnum;
        }livro;

livro*  aloca(livro *p, int tam);                    //retorna o endereço da memória alocada

main()
{
livro    *ptr=NULL;

ptr = aloca(ptr, 10);                                //chamada por valor
}//main

livro*  aloca(livro *p, int tam)
{
if((p=(livro*)realloc(p, tam*sizeof(livro)))== NULL)
  {
   printf("Erro de alocacao");
   exit(1);
  }
return p;
}//aloca
```

2.6 - Ponteiros como Membros de Estruturas

Esses ponteiros são declarados exatamente como os ponteiros que não pertencem a qualquer estrutura - ou seja, usando o operador de indireção (*).

Ex.:

```
        struct dados {
                int     *valor;
                int     *taxa;
                } primeira;
```

Estas instruções definem e declaram uma estrutura cujos dois membros são ponteiros para o tipo int. Como acontece com os ponteiros em geral, não basta declará-los; devemos também inicializá-los fazendo com que eles apontem para o endereço de uma variável.

Ex.- custo e juros são declaradas como variáveis do tipo int:

```
primeira.valor  =  &custo;
primeira.taxa   =  &juros;
```

Agora que os ponteiros já foram inicializados, podemos usar o operador de indireção (*).

a expressão ***primeira.valor** é avaliada como o valor de custo
a expressão ***primeira.taxa** é avaliada como o valor de juros.

Ex.:

```
struct msg {
        char    *p1;
        char    *p2;
} apostila;

apostila.pl = "Estruturas em C";
apostila.p2 = "Profª Andrea";
```

Os ponteiros como membros de estruturas podem ser usados em qualquer lugar em que um ponteiro normal poderia ser usado. Por exemplo, para imprimir as strings para os quais os ponteiros estão apontando, podemos escrever:

```
printf("%s %s", apostila.pl, apostila.p2);
```

Podemos utilizar uma matriz do tipo char ou um ponteiro como membros de uma estrutura. Ambos são usados para armazenar um string em uma estrutura.

Ex. - a estrutura msg usa ambos os métodos

```
struct msg {
        char    p1[16];
        char    *p2;
} apostila;
```

Podemos usar estes dois membros da estrutura de maneira semelhante:

```
strcpy(apostila.pl, "Estruturas em C");
strcpy(apostila.p2, "Profª Andrea");
                ou
puts (apostila.pl);
puts (apostila.p2);
```

Qual é a diferença entre esses dois métodos? É a seguinte, se definirmos uma estrutura que contenha uma matriz do tipo char, todas as instâncias desse tipo de estrutura conterão espaço de armazenagem para uma matriz do tamanho especificado. Além disso, estaremos limitado ao tamanho especificado e não poderemos armazenar um string maior na estrutura.

Ex.:

```
struct msg {
        char    p1[16];
        char    p2[13];
} apostila;

strcpy(apostila.p1, "Estruturas e Ponteiros em C");        // erro - string é mais longo que a matriz
strcpy(apostila.p2, "ALB");                                // desperdiça espaço, string menor que a matriz
```

Se, por outro lado, definirmos uma estrutura que contenha ponteiros para o tipo char, essas restrições não se aplicam. Cada instância da estrutura conterá somente o espaço de armazenagem necessário para o próprio ponteiro. **Os strings propriamente ditos serão armazenados em qualquer outra parte da memória** e teremos que utilizar alocação dinâmica para armazenar esses strings.

Dessa forma, não há qualquer restrição ao seu tamanho, nem espaço ocioso, pois os strings não são armazenados como parte da estrutura. Cada ponteiro da estrutura pode apontar para um string de qualquer tamanho. **Embora não sejam armazenados na estrutura, os strings referenciados pelos ponteiros passam a fazer parte da estrutura que contém esses ponteiros**.

Ex.:

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

typedef struct msg {
char    *p1;
char    *p2;
} msg;

main( )
{
msg apostila;                   //estrutura msg chamada apostila
char aux[80];                   // string auxiliar
int tam;

printf("Titulo da apostila: ");
gets(aux);
tam=strlen(aux);
if((apostila.p1=(char*)realloc(NULL,tam*sizeof(char)))==NULL)        //aloca espaço p/o 1º membro
  {
   printf("\nErro de alocacao\n");
   exit(1);
   }
strcpy(apostila.p1,aux);
printf("Autor da apostila: ");
gets(aux);
tam=strlen(aux);
```

```
if((apostila.p2=(char*)realloc(NULL,tam*sizeof(char)))==NULL)          //aloca espaço p/o 2º membro
   {
    printf("\nErro de alocacao\n");
    exit(1);
   }
strcpy(apostila.p2,aux);
printf("%s %s",apostila.p1,apostila.p2);
}//main
```

2.7 - Ponteiros como Membros de Estruturas e Ponteiros para Estruturas

Agora já podemos refazer o exemplo anterior, utilizando ponteiros como membros de estruturas e que apontem para estruturas, como mostrado a seguir:

Diferentemente do exercício anterior, iremos declarar um ponteiro para a estrutura (*p_apostila) e não mais uma variável do tipo estrutura (apostila), então, primeiro deveremos alocar espaço para a estrutura e só depois alocar espaço para cada membro dessa estrutura.

Outra diferença: como estamos trabalhando com ponteiro para estrutura, devemos utilizar o operador de acesso indireto (**->** p_apostila->p1) e não mais o operador de associação ou ponto (**.** apostila.p1) que é utilizado com variáveis simples.

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

typedef struct msg {
char    *p1;
char    *p2;
} msg;

main( )
{
msg *p_apostila=NULL;              //ponteiro para estrutura msg chamada p_apostila
char aux[80];                      // string auxiliar
int tam;

if((p_apostila=(msg*)realloc(NULL,sizeof(msg)))==NULL)        //alocação dinâmica da estrutura
   {
    printf("\nErro de alocacao\n");
    exit(1);
   }
printf("Titulo da apostila: ");
gets(aux);
tam=strlen(aux);
if((p_apostila->p1=(char*)realloc(NULL,tam*sizeof(char)))==NULL)    //aloca espaço p/o 1º membro
   {
    printf("\nErro de alocacao\n");
    exit(1);
   }
strcpy(p_apostila->p1,aux);
printf("Autor da apostila: ");
```

```
gets(aux);
tam=strlen(aux);
if((p_apostila->p2=(char*)realloc(NULL,tam*sizeof(char)))==NULL)     //aloca espaço p/o 2º membro
   {
    printf("\nErro de alocacao\n");
    exit(1);
   }
strcpy(p_apostila->p2,aux);
printf("%s %s",p_apostila->p1,p_apostila->p2);
}//main
```

2.8 – Exercícios Propostos

1. Dado a estrutura abaixo, implemente uma rotina de cadastro, deve-se consultar o usuário para continuar. O registro deve ser gerado automaticamente pelo sistema. **Utilizar alocação dinâmica e ponteiros para a estrutura.**

```
struct agenda
{
      int reg;
      char nome[80];
      float nota;
 };
```

2. Fazer um programa para **Sistema de Conta Bancária** – este programa se destina a controlar as contas de clientes. Consultar o usuário para continuar. **Utilizar alocação dinâmica e ponteiros para a estrutura.**

[1] Cadastro - receber os valores digitados pelo usuário. Apenas um registro é cadastrado por vez.

[2] Depósito - o acesso deve ser feito através do nº de conta corrente. Buscar o registro, mostrar o nome do cliente e o saldo para simples conferência, pedir o valor do depósito, fazer as alterações e apresentar na tela o saldo atualizado.

[3] Retirada - o acesso deve ser feito através do nº de conta corrente. Buscar o registro, mostrar o nome do cliente e o saldo para simples conferência, pedir o valor da retirada, fazer as alterações se possível (a retirada só será permitida, se houver saldo suficiente) e apresentar na tela o saldo atualizado.

```
struct cliente{
        char nome[30];
        int conta;
        float saldo;
};
```

3. Fazer um programa para **Diário Eletrônico** – este programa se destina a controlar as notas e a média dos alunos. Consultar o usuário para continuar. **Utilizar alocação dinâmica e ponteiros para a estrutura.**

[1] Cadastro - receber os valores digitados pelo usuário, inicialmente notas e média=0. Apenas um registro é cadastrado por vez.

[2] Controle de Notas - o acesso deve ser feito através do RA. Buscar o registro, mostrar o nome do aluno para simples conferência, fazer as alterações das notas, calcular a média e apresentar na tela as notas e a média.

```
struct aluno{
        char nome[80];
        char RA[79];
        float nota[2];          //notas de provas – considerar 2 provas
        float media;            //média aritmética das provas
};
```

4. Fazer um programa para **Controle de Hotel** - este programa se destina a controlar o check-in (cadastro de hóspedes) de um hotel. O hotel possui 15 quartos. **Utilizar alocação dinâmica e ponteiros para a estrutura.**

[1] Check-in - alocar dinamicamente espaço, receber os valores digitados pelo usuário, se o hóspede não tiver acompanhantes atribuir categoria **S**olteiro, caso contrário **F**amiliar, buscar o número do quarto disponível, de acordo com a categoria na estrutura **quartos.** Apenas um hóspede é cadastrado por vez. Não esquecer de atualizar o quarto da estrutura **quartos** para **O**cupado.

[2] Check-out – encerra a estadia e apresenta o relatório, de acordo com o quarto. Apenas um registro é acessado por vez, buscar e mostrar o número do quarto, o nome do hóspede, quantidade de acompanhantes, a categoria (Solteiro ou Familiar,o tempo de permanência em dias e o valor a ser pago.

[3] Fim

Dica:

- No check-in - não esquecer de verificar se na estrutura **hospede** há um espaço vago (cujo quarto = -1), se houver o novo hóspede deverá ser ali armazenado, caso contrário, acrescentar no final da estrutura.

```
struct hospede{
        int quarto;             // número do quarto
        char nome[80];
        int acompanhante;       // quantidade de acompanhantes
        char categoria;         // [S]olteiro / [F]amiliar
        int dias;               // tempo de permanência – em dias
};

struct quarto{
        int num;                // número do quarto
        char categoria          // [S]olteiro / [F]amiliar
        char status             // [L]ivre / [O]cupado
};
```

Categoria de quarto:
[**S**]olteiro – diária R$ 85,00 por pessoa
[**F**]amiliar – diária R$ 45,00 por pessoa

5. Aloque dinamicamente espaço e receba a seqüência abaixo, até que o usuário escolha a opção **N** (Deseja continuar? (S/**N**)). **Utilizar ponteiros para a estrutura e ponteiros como membros da estrutura.**

```
struct aluno{
    char *nome;          //aponta para o nome
    char *curso;         //aponta para o curso
    };
```

Cursos: Civil, Computação, Elétrica, Mecânica, Mecatrônica, Química ou Produção

6. Idem ao anterior, porém como os nomes e os cursos se repetem, verificar se o nome ou o curso já existem, se sim, basta apontar para o endereço já existente, se não, iremos alocar espaço e cadastrar o novo nome ou curso. **Utilizar ponteiros para a estrutura e ponteiros como membros da estrutura.**

```
struct aluno{
    char *nome;          //aponta para o nome ou p/ o endereço que contém o nome
    char *curso;         //aponta para o curso ou p/ o endereço que contém o curso
    };
```

Cursos: Civil, Computação, Elétrica, Mecânica, Mecatrônica, Química ou Produção

7. Idem ao anterior, porém como os nomes são concentrados em poucos cursos, iremos criar uma "Lista de Cursos" para facilitar nossa busca. A Lista deverá ser alocada dinamicamente à medida que vai sendo formada**. Utilizar ponteiros para a estrutura e ponteiros como membros da estrutura e deve ser criado um ponteiro para um vetor de ponteiros (Lista de Cursos).**

```
struct aluno{
    char *nome;          // aponta para o nome ou p/ o endereço que contém o nome
    char *curso;         //aponta para o endereço do curso
    };
```

Cursos: Civil, Computação, Elétrica, Mecânica, Mecatrônica, Química ou Produção

8. Idem ao anterior, porém permitiremos Alterar os Cursos da "Lista de Cursos", ou seja, o aluno poderá trocar de curso. Não esquecer de verificar se o novo curso já aparece na lista, se sim, basta apontar para o novo endereço, se não, ele deve ser cadastrado na "Lista de Cursos" (verificar se há espaço vago na lista, cujo conteúdo = NULL, caso contrário deverá ser alocado espaço na lista e na memória). Caso não sobre mais nenhum nome daquele curso na Lista, o mesmo deverá ser descartado, ou seja, conteúdo recebe NULL. **Utilizar ponteiros para a estrutura e ponteiros como membros da estrutura e deve ser criado um ponteiro para um vetor de ponteiros (Lista de Cursos).**

```
struct aluno{
    char *nome;          // aponta para o nome ou p/ o endereço que contém o nome
    char *curso;         //aponta para o endereço do curso
    };
```

Cursos: Civil, Computação, Elétrica, Mecânica, Mecatrônica, Química ou Produção

3. UNIÕES

3.1 - Introdução

As uniões são semelhantes às estruturas. Uma união é declarada e usada exatamente como uma estrutura. A única diferença está no fato de que somente **um** dos seus membros pode ser usado de cada vez. A razão para isso é simples:

todos os membros de uma união ocupam uma mesma área da memória. Eles são dispostos um sobre o outro.

Estruturas e uniões são localizações de memória usadas para agrupar um número de variáveis de tipos diferentes juntas, mas, enquanto os membros de uma estrutura são armazenados em espaços diferentes de memória, membros de uma união compartilham da mesma localização de memória.

Em outras palavras, a união é uma forma de tratamento de uma seção de memória contendo um tipo de variável numa ocasião e um outro tipo de variável numa outra ocasião.

quando você declara uma união é automaticamente alocado espaço de memória para conter o membro de maior tipo de sua lista de membros.

3.2 – Definindo, Declarando e Inicializando Uniões

As uniões são definidas e declaradas exatamente como as estruturas. A única diferença na declaração é o uso da palavra-chave **union** ao invés de **struct**.

sintaxe:

```
union  exemplo {
        char    inicial;
        int     idade;
        float   salário;
        }pessoal;
```

Essa união pode ser usada para criar instâncias que contenha um valor de caractere - inicial, ou um valor inteiro - idade, ou um valor de ponto flutuante - salário. Esta é uma condição **OR**; ao contrário de uma estrutura, que pode conter ambos os valores, a união só pode conter um valor de cada vez.

Declaramos uma variável de nome pessoal do tipo union exemplo e para esta variável foram reservados 4 bytes de memória tendo em vista que o maior de seus membros é do tipo float. Este espaço não é grande o bastante para armazenar todos os membros ao mesmo tempo, mas suficiente para guardar qualquer um dos membros.

Uma união pode ser inicializada ao ser declarada. Como somente um membro pode ser usado de cada vez, somente um poderá ser inicializado. Para evitar confusão, somente o **primeiro** membro de uma união pode ser inicializado.

Portanto, no exemplo acima só conseguiremos inicializar a variável do tipo char: pessoal.inicial. E se quisermos inicializar a variável do tipo float: pessoal.salário, devemos inverter a ordem, vindo a mesma em 1º lugar. Na falta de inicialização explícita, cada membro é inicializado com zero.

Ex.:

```
union  exemplo {
        float   salário;                    // deve ser declarado primeiro
        char    inicial;
        int     idade;
        }pessoal  =  {123.45};
```

3.3 – Acessando Membros da União

O operador ponto (.) é usado para acessar membros da união. Devemos observar que o endereço de memória é o mesmo para qualquer membro. Assim não podemos ter o valor de mais de um membro ao mesmo tempo.

Se tentarmos referenciar o valor de pessoal.idade depois de atribuirmos um valor a pessoal.salário obteremos algo sem sentido, porque o programa interpretará como inteiro os dois primeiros bytes da variável float lá guardada.

o programador é responsável pela preservação da seqüência de acesso e armazenamento de dados dos membros de uma união.

Ex.:

```
main( )
{
        union  exemplo{
                char    inicial;
                int     idade;
                float   salario;
                };

union exemplo         pessoal;
printf("sizeof(union exemplo) = %i \n", sizeof(union exemplo));
pessoal.inicial = 'A';
printf("inicial = %c            endereço = %u \n", pessoal.inicial, &pessoal.inicial);
pessoal.idade = 35;
printf("idade = %i              endereço = %u \n", pessoal.idade, &pessoal.idade);
pessoal.salario = 850;
printf("salario = %.2f          endereço = %u \n", pessoal.salario, &pessoal.salario);
}
```

A saída será:

```
sizeof(union exemplo)       =       4
inicial = A             endereço=65492
idade = 35              endereço=65492
salário = 850           endereço=65492
```

3.4 - Por que usar Uniões?

Uma das razões é a possibilidade de usar um único nome para dados de diferentes tipos. Podemos reescrever a função de biblioteca C **sqrt()**, por exemplo.

A função original aceita somente um argumento do tipo **double** e poderá aceitar qualquer tipo de dado. A definição da união será:

Ex.:

```
union num {
        double      dnum;
        float       fnum;
        long        lnum;
        int         inum;
        char        cnum;
        };
```

Assim a mesma função servirá para todos os tipos de dados. (A função deverá examinar os dados da união para determinar de que tipo é o argumento ou receber a informação via um segundo argumento.)

Uma união pode ser realmente conveniente quando queremos que um mesmo dado seja tratado com tipos diferentes, dependendo das necessidades da função que o usa.

A união fornece uma forma de ver um mesmo dado de várias diferentes maneiras.

3.5 - Uniões de Estruturas

Exatamente como uma estrutura pode ser membro de outra estrutura, uniões podem ser membros de outra união, uniões podem ser membros de estruturas e estruturas podem ser membros de uniões.

Ex.:

```
main()
{
        struct  doisint {
                int     intnuml;
                int     intnum2;
                };

        union   intflo {
                struct doisint  num;
                float   fltnum;
                }unex;

printf("sizeof(union intflo)=%i \n", sizeof(union intflo));
unex.num.intnuml = 734;
unex.num.intnum2 = -333;
printf("unex.num.intnuml=%i \n", unex.num.intnuml);
printf("unex.num.intnum2=%i \n",unex.num.intnum2);
unex.fltnum=867.43;
printf("unex.fltnum=%.1f \n",unex.flrnum);
}
```

A saída será :

```
sizeof(unionintflo) = 4
unex.num.intnum1= 734
unex.num.intnum2 =-333
unex.fltnum = 867.4
```

Neste exemplo declaramos um tipo estrutura doisint e uma variável estrutura num deste tipo. Os membros da estrutura são dois inteiros, intnuml e intnum2. Declaramos também um tipo união intflo e uma variável união unex deste tipo. Os membros da união são: a estrutura num e a variável do tipo float fltnum.

Um membro da estrutura da união é acessado usando o operador ponto duas vezes:

unex.num.intnum1

3.6 – Exercícios Propostos

1. Refazer o programa para **Controle de Hotel** utilizando união - este programa se destina a controlar o check-in (cadastro de hóspedes) de um hotel. O hotel possui 15 quartos. **Utilizar alocação dinâmica, ponteiros para a estrutura e para a união.**

[1] Check-in - alocar dinamicamente espaço, receber os valores digitados pelo usuário, se o hóspede não tiver acompanhantes atribuir categoria **S**olteiro, caso contrário **F**amiliar, buscar o número do quarto disponível, de acordo com a categoria na estrutura **quartos.** Apenas um hóspede é cadastrado por vez. Não esquecer de atualizar o quarto da estrutura **quartos** para **O**cupado.

[2] Check-out – encerra a estadia e apresenta o relatório, de acordo com o quarto. Apenas um registro é acessado por vez, buscar e mostrar o número do quarto, o nome do hóspede, quantidade de acompanhantes, a categoria (Solteiro ou Familiar,o tempo de permanência em dias e o valor a ser pago.

[3] Fim

<u>Dicas</u>:

- **Utilizar ponteiros**
- No chech-in - não esquecer de verificar se na estrutura **hospede** há um espaço vago (cujo quarto = -1), se houver o novo hóspede deverá ser ali armazenado, caso contrário, acrescentar no final da estrutura.

```
struct composto{
char    sigla;                  // [O]cupado
int     num;                    // número do quarto ou do registro do hóspede
};
union   estado{
char    sigla;                  // [L]ivre
struct composto     campo;
};
```

```
struct hospede{
        int numreg;                 // número do registro do hóspede
        char nome[80];
        int acompanhante;           // quantidade de acompanhantes
        int dias;                   // tempo de permanência – em dias
        union estado   tabela;
};

struct quarto{
        int num;                    // número do quarto
        char categoria              // [S]olteiro / [F]amiliar
        char status                 // [L]ivre / [O]cupado
        union estado   status;
};
```

Categoria de quarto:
[**S**]olteiro – diária R$ 85,00 por pessoa
[**F**]amiliar – diária R$ 45,00 por pessoa

Exemplo do Cadastro de Hóspedes

Nº Registro	Nome	Acompanhante	Dias	Tabela
100	Ana Teresa dos Santos	1	7	L
101	Beto Leme	0	10	O - 001
102	Caio Alves Lima	4	5	O - 004

Exemplo do Cadastro de Quartos

Nº quarto	Categoria	Status
001	S	O - 101
002	F	L
003	S	L
004	F	O - 102