INSTITUTO FEDERAL
RIO GRANDE DO SUL

Algoritmos e Programação II

Ponteiros

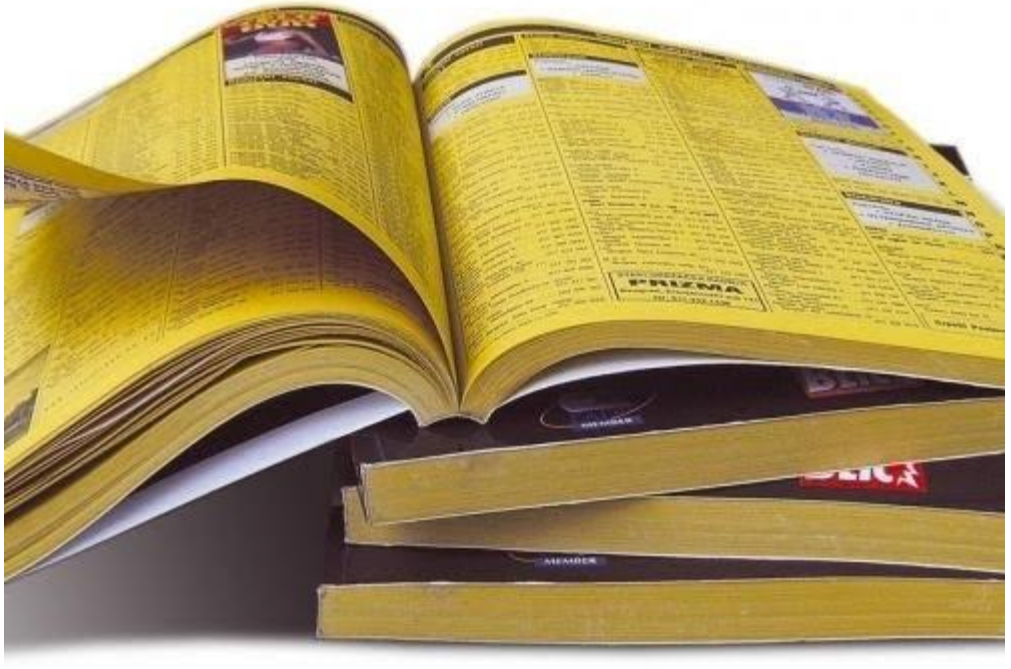
```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Referência

- Referência é quando nos referimos diretamente ao identificador do endereço da memória.
- A memória de um computador é na verdade uma grande tabela com células seqüenciais, cada célula tem seu próprio endereço que segue um padrão contínuo. Ou seja, a primeira célula será 0x00000000, a segunda 0x00000001, a terceira 0x00000002, e assim por diante.
- Quando fazemos referência, estamos obtendo exatamente este valor, que é o endereço da célula na memória.
- A referência é dada pelo operador &.

Dereferência

- Dereferência é quando nos referimos ao valor contido no endereço armazenado, ou seja, é o contrário da operação de referência.



Operações com Referência/Deferência

Quando incrementamos um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta. O decremento funciona semelhantemente. Supondo que **p** é um ponteiro, as operações são escritas como:

```
p++;        p--;
```

Por exemplo, para incrementar o conteúdo da variável apontada pelo ponteiro **p**, faz-se:

```
(*p)++;
```

Outras operações aritméticas úteis são a soma e subtração de inteiros com ponteiros. Para incrementar um ponteiro de 15 (o destino do ponteiro). Basta fazer:

```
p=p+15;   ou   p+=15;
```

E se você quiser usar o conteúdo do ponteiro 15 posições adiante:

```
*(p+15);
```

A subtração funciona da mesma maneira.

Vetores como ponteiros

Na declaração da matriz: **int mat[10][10]** o compilador C calcula o tamanho, em bytes, necessário para armazenar esta matriz.

Este tamanho é: **tam1 + tam2 + tam3 + ... + tamN * tamanho_do_tipo**

O compilador aloca este número de bytes em um espaço livre de memória. O ***nome da variável*** que você declarou é na verdade *um ponteiro para o tipo da variável da matriz*.

Durante a alocação na memória o espaço para a matriz, o nome da variável (que é um ponteiro) e aponta para o *primeiro* elemento da matriz.

Então como é que podemos usar a seguinte notação?

mat[5]

o que é igual a

***(mat+5)**

Vetores como ponteiros

Todo vetor começa com indexação zero É porque, ao pegarmos o valor do primeiro elemento de um vetor, queremos, de fato, ***nome_da_variável** e então devemos ter um índice igual a zero.

Assim sabemos que:

*nome_da_variável é equivalente a nome_da_variável[0]

Podemos ter índices negativos. Estaríamos pegando posições de memória antes do vetor. O compilador C não verifica a validade dos índices. Ele *não* sabe o tamanho do vetor. Ele apenas aloca a memória, ajusta o ponteiro do nome do vetor para o início do mesmo e, quando você usa os índices, encontra os elementos requisitados.

VARREDURA SEQUENCIAL DE UMA MATRIZ

Quando temos que varrer todos os elementos de uma matriz de uma forma sequencial, podemos usar um ponteiro, o qual vamos incrementando.

Qual a vantagem?

Considere o seguinte programa para zerar uma matriz:

```
int main ()
{           float matrx [50][50];
int i,j;
            for (i=0;i<50;i++)
                        for (j=0;j<50;j++)
                                    matrx[i][j]=0.0;
            return(0);
}
```

O programa, cada vez que se faz **matrx[i][j]** o programa tem que calcular o deslocamento para dar ao ponteiro. Ou seja, o programa tem que calcular 2500 deslocamentos.

VARREDURA SEQUENCIAL DE UMA MATRIZ

Podemos reescrevê-lo usando ponteiros:

```
int main ()
{
      float matrx [50][50];
      float *p;
      int count;
      p=matrx[0];
      for (count=0;count<2500;count++)
     {
      *p=0.0;
                p++;
      }
      return(0);
}
```

No segundo programa o único cálculo que deve ser feito é o de um incremento de ponteiro. Fazer 2500 incrementos em um ponteiro é muito mais rápido que calcular 2500 deslocamentos completos.

OBS.: um ponteiro é uma variável, mas o nome de um vetor não é uma variável.

Isto significa, que não se consegue alterar o endereço que é apontado pelo "nome do vetor".

PONTEIROS COMO VETORES

Considerando que o nome de um vetor é um ponteiro constante e, também podemos indexar o nome de um vetor, como conseqüência podemos também indexar um ponteiro qualquer.

```
#include <stdio.h>
int main ()
{
    int matrx [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int *p;
    p=matrx;
    printf ("O terceiro elemento do vetor e: %d",p[2]);
    return(0);
}
```

Podemos ver que **p[2]** equivale a ***(p+2)**.

PASSANDO VETORES COMO ARGUMENTOS DE FUNÇÕES

Os ponteiros podem ser passados como argumentos de funções

```
#include <stdio.h>
void atribuiValores(int[], int);
void mostraValores(int[], int);
int main()
{
int vetorTeste[3];
atribuiValores(vetorTeste, 3);
mostraValores(vetorTeste, 3);
return 0;
}

void atribuiValores(int valores[], int num)
{   for (int i = 0; i < num; i++)
        {     printf("Insira valor %d: ", i + 1);
        scanf("%d", &valores[i]);
        }
}
void mostraValores(int valores[], int num)
{   for (int i = 0; i < num; i++)
 {
        printf("Valor #%d: %d\n", i + 1, valores[i]);
   }
}
```

Repare que passamos dois parâmetros para as funções:

O "nome" do vetor, que representa o seu endereço na memória.

Temos 3 maneiras para passar o endereço do vetor:

1) Diretamente pelo seu "nome“.

2) Via um ponteiro.

3) Endereço do primeiro elemento.

3 MANEIRAS PARA PASSAR O ENDEREÇO DO VETOR

//Passando como ponteiro

```
#include <stdio.h>
#include <stdlib.h>
void atribuiValores(int valores[], int num)
{   for (int i = 0; i < num; i++)
        {     printf("Insira valor %d: ", i + 1);
        scanf("%d", &valores[i]);
        }
}
void mostraValores(int valores[], int num)
{   for (int i = 0; i < num; i++)
 {      printf("Valor #%d: %d\n", i + 1, valores[i]);
   } }

int main()
{
int vetorTeste[3];
int * pont = vetorTeste;
atribuiValores(pont, 3);
mostraValores(pont, 3);
system("pause");
return 0;
}
```

3 MANEIRAS PARA PASSAR O ENDEREÇO DO VETOR

//Passando o endereço da primeira posição

```
#include <stdio.h>
#include <stdlib.h>
void atribuiValores(int valores[], int num)
{   for (int i = 0; i < num; i++)
        {     printf("Insira valor %d: ", i + 1);
        scanf("%d", &valores[i]);
        } }
void mostraValores(int valores[], int num)
{   for (int i = 0; i < num; i++)
 {
        printf("Valor #%d: %d\n", i + 1, valores[i]);
   } }
int main()
{
int vetorTeste[3];

atribuiValores(&vetorTeste[0], 3);
mostraValores(&vetorTeste[0], 3);
system("pause");
return 0;
}
```

MANEIRAS PARA PASSAR O ENDEREÇO DO VETOR

//Passando o endereço pelo nome do vetor

```
#include <stdio.h>
#include <stdlib.h>
void atribuiValores(int valores[], int num)
{   for (int i = 0; i < num; i++)
        {     printf("Insira valor %d: ", i + 1);
        scanf("%d", &valores[i]);
        }}

void mostraValores(int valores[], int num)
{   for (int i = 0; i < num; i++)
 {
        printf("Valor #%d: %d\n", i + 1, valores[i]);
   }}

int main()
{
int vetorTeste[3];
atribuiValores(vetorTeste, 3);
mostraValores(vetorTeste, 3);
system("pause");
return 0;
}
```

Ponteiros para Ponteiros

Podemos declarar um ponteiro para um ponteiro com a seguinte notação:

*tipo_da_variável **nome_da_variável;*

**nome_da_variável é o conteúdo final da variável apontada;

*nome_da_variável é o conteúdo do ponteiro intermediário.

No C podemos declarar ponteiros para ponteiros para ponteiros, ou então, ponteiros para ponteiros para ponteiros para ponteiros e assim por adiante.

Para acessar o valor desejado apontado por um ponteiro para ponteiro, o operador asterisco deve ser aplicado duas vezes.

```
#include <stdio.h>
    int main()
    {
       float fpi = 3.1415, *pf, **ppf;
       pf = &fpi;              /* pf armazena o endereco de fpi */
       ppf = &pf;               /* ppf armazena o endereco de pf */
       printf("%f", **ppf);     /* Imprime o valor de fpi */
       printf("%f", *pf);       /* Tambem imprime o valor de fpi */
       return(0);
    }
```

INSTITUTO FEDERAL
RIO GRANDE DO SUL

Algoritmos e Programação II

Alocação Dinâmica

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

ALOCAÇÃO DINÂMICA

A alocação dinâmica permite alocar memória em tempo de execução, ou seja, alocar memória para novas variáveis ou redimensionar as existentes quando o programa está sendo executado.

O padrão C ANSI define apenas 4 funções para o sistema de alocação dinâmica, disponíveis na biblioteca **stdlib.h**:

- **malloc**
- **calloc**
- **realloc**
- **free**

malloc

A função **malloc()** serve para alocar memória e tem o seguinte protótipo:

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

A funçao toma o número de bytes que queremos alocar (**num**), aloca na memória e retorna um ponteiro **void *** para o primeiro byte alocado. O ponteiro **void *** pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função **malloc()** retorna um ponteiro nulo.

malloc

Veja um exemplo de alocação dinâmica com malloc():

```
main (void)
{
 int *p;
 float a = 100000000;   /* Determina o valor de a */
       p= (int *)malloc (a * sizeof(float));
       if (!p)
        {
        printf ("** Erro: Memoria Insuficiente **");
        }
        else
        {
        printf ("Memória alocada som sucesso!!!");
        }
        system("pause");
return 0;
}
```

malloc

No exemplo acima, é alocada memória suficiente para se colocar **a** números float.

O operador **sizeof()** retorna o número de bytes de um inteiro.

Ele é util para se saber o tamanho de tipos. O ponteiro **void*** que **malloc()** retorna é convertido para um **int*** pelo cast e é atribuído a **p**. A declaração seguinte testa se a operação foi bem sucedida. Se não tiver sido, **p** terá um valor nulo, o que fará com que **!p** retorne verdadeiro. Se a operação tiver sido bem sucedida, podemos usar o vetor de inteiros alocados normalmente, por exemplo, indexando-o de **p[0]** a **p[(a-1)]**.

calloc

A função **calloc()** também serve para alocar memória, mas possui um protótipo um pouco diferente:

```
void *calloc (unsigned int num, unsigned int size);
```

A função aloca uma quantidade de memória igual a **num * size**, isto é, aloca memória suficiente para uma matriz de **num** objetos de tamanho **size**. Retorna um ponteiro **void *** para o primeiro byte alocado.

O ponteiro **void *** pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função **calloc()** retorna um ponteiro nulo.

realloc

A função **realloc()** serve para realocar memória e tem o seguinte protótipo:

```
void *realloc (void *ptr, unsigned int num);
```

A função modifica o tamanho da memória previamente alocada apontada por ***ptr** para aquele especificado por **num**.

O valor de **num** pode ser maior ou menor que o original. Um ponteiro para o bloco é devolvido porque **realloc()** pode precisar mover o bloco para aumentar seu tamanho. Se isso ocorrer, o conteúdo do bloco antigo é copiado no novo bloco, e nenhuma informação é perdida. Se **ptr** for nulo, aloca **size** bytes e devolve um ponteiro; se **size** é zero, a memória apontada por **ptr** é liberada. Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.

free

Quando alocamos memória dinamicamente é necessário que nós a liberemos quando ela não for mais necessária. Para isto existe a função **free()** cujo protótipo é:

```
void free (void *p);
```

Basta então passar para **free()** o ponteiro que aponta para o início da memória alocada.

INSTITUTO FEDERAL
RIO GRANDE DO SUL

Algoritmos e Programação II

Alocação de Memória
Tipos de Dados
Dados Primitivos
Tipos de Dados Abstratos

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Alocação de memória

Consiste no processo de solicitar/utilizar memória durante o processo de execução de uma aplicação. A alocação de memória no computador pode ser dividida em dois grupos principais:

Alocação Estática

Alocação Dinâmica

Alocação de memória

Alocação Estática: os dados tem um tamanho fixo e estão organizados seqüencialmente na memória do computador. Um exemplo típico de alocação estática são as variáveis globais e arrays;

Alocação de memória

Alocação Dinâmica: os dados não precisam ter um tamanho fixo, pois podemos definir para cada dado quanto de memória que desejamos usar. Sendo assim vamos alocar espaços de memória (blocos) que não precisam estar necessariamente organizados de maneira seqüencial, podendo estar distribuídos de forma esparsa na memória do computador.

Na alocação dinâmica, vamos pedir para alocar/desalocar blocos de memória, de acordo com a nossa necessidade, reservando ou liberando blocos de memória durante a execução de um programa. Para poder “achar” os blocos esparsos na memória usamos as variáveis do tipo Ponteiro (indicadores de endereços de memória).

Relembrando... Sessão revisão

SISTEMA BINÁRIO

Como o próprio nome já indica, tem base 2 e é o sistema de numeração mais utilizado em processamento de dados digital, pois utiliza apenas dois símbolos ou algarismos 0 e 1.

Também vale ressaltar, que em processamentos digitais, que o dígito 1 também é conhecido por nível lógico 1, nível lógico alto, ligado, verdadeiro e energizado. Já o dígito 0 poder ser nível lógico 0, nível lógico baixo, desligado, falso e desernegizado.

Relembrando... Sessão revisão

Words

O termo word refere-se a um grupo de bits processados simultaneamente por processadores de uma determinada arquitetura.

Portanto, sua largura depende do processador em questão. Várias diferentes larguras são usadas, incluindo 6, 8, 12, 16, 18, 24, 32, 36, 39, 48, 60 e 64 bits. Atualmente a largura de 32 bits é a mais comum entre computadores de uso geral, com a chegada cada vez mais comum de larguras de 64 bits.

Relembrando... Sessão revisão

Exemplos comuns

Largura (bits)	Nome	Bloco com sinal	Bloco sem sinal	Uso
8	byte, octeto	-128 a +127	0 a +255	Caracteres ASCII, C int8_t, Java byte
16	*halfword*, *word*	-32 768 a +32 767	0 a +65 535	Caracteres UCS-2, C int16_t, Java char, Java short
32	*word*, *doubleword*, *longword*	-2 147 483 648 a +2 147 483 647	0 a +4 294 967 295	Caracteres UCS-4, Truecolor com canal alfa, C int32_t, Java int
64	*doubleword*, *longword*, *quadword*	-9 223 372 036 854 775 808 a +9 223 372 036 854 775 807	0 a +18 446 744 073 709 551 615	C int64_t, Java long
128		-170 141 183 460 469 231 731 687 303 715 884 105 728 a +170 141 183 460 469 231 731 687 303 715 884 105 727	0 a +340 282 366 920 938 463 463 374 607 431 768 211 455	

Relembrando... Sessão revisão

Por que os caracteres consomem 1 byte?

Todos os caracteres representáveis na memória foram codificados em uma tabela denominada Tabela ASCII (American Standard Code InterchangeInformation).

Originalmente a Tabela ASCII possuía apenas 128 códigos que representavam o alfabeto, os caracteres de pontuação e alguns caracteres de controle do idioma inglês. Atualmente, além dos 128 códigos originais, possui mais 128 códigos adicionais para um conjunto de caracteres denominado caracteres estendidos.

Assim, cada tecla do teclado tem um código ASCII associado a ela, de modo que, quando uma tecla é digitada, uma posição de memória ou 1 B da memória será preenchido com o seu código correspondente.

Relembrando... Sessão revisão

Por que um inteiro ocupa 4 bytes?

Da mesma forma, o primeiro bit é usado para o sinal, e os 31 bits restantes para armazenar o valor do número na base dois.

[s][ ][ ][ ][ ][ ][ ][ ] [ ][ ][ ][ ][ ][ ][ ] [ ][ ][ ][ ][ ][ ][ ] [ ][ ][ ][ ][ ][ ][ ] [ ][ ][ ][ ][ ][ ][ ]

Limitação: Neste caso, o valor do número inteiro será armazenado usando 31 bits. Portanto:

o maior inteiro longo será 2 a 31 -1 = + 2.147.483.647

o menor inteiro longo será -2 a 31 = - 2.147.483.648

Relembrando... Sessão revisão

Alocação dinâmica de memória - Visão mais detalhada da memória do computador

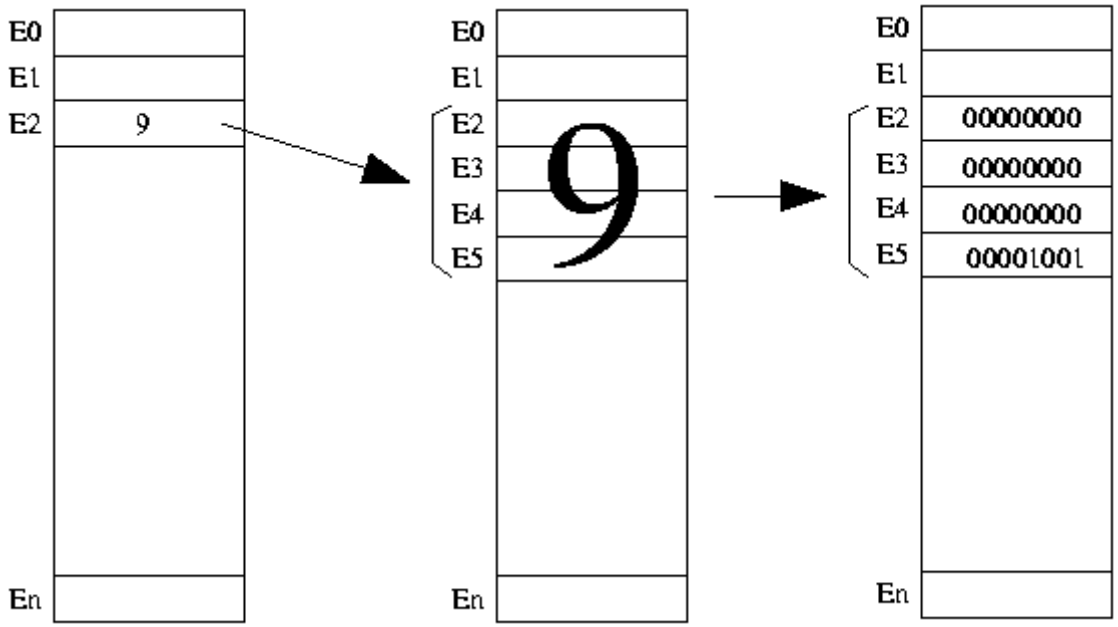
Considere o caso abaixo:

```
int x = 9;
```

O x poderia ir parar à posição de memória E2 quando o programa fosse executado. Na realidade, as coisas não são bem assim. Cada posição de memória corresponde apenas a um byte e uma variável inteira ocupa geralmente 4 bytes. Ou seja, o número 9 não vai estar representado numa única posição de memória mas sim em 4 posições de memória (ex: E2, E3, E4 e E5). Aliás, aquilo que está nessas 4 posições de memória é a representação em binário do número 9.

Relembrando... Sessão revisão

Apesar da variável x estar localizada nos endereços E2, E3, E4 e E5, costuma dizer-se que o endereço da variável x é E2. Isto é, o endereço de uma variável é o endereço do primeiro byte que a variável ocupa.



Relembrando... Sessão revisão

E os arrays? Como é que são guardados?

Se declararmos um array a de 10 inteiros, o compilador vai reservar um bloco de memória consecutivo que permita guardar esses 10 inteiros. Se um inteiro ocupar 4 bytes, o compilador terá de reservar um bloco de 40 bytes (por exemplo, do endereço E100 até ao endereço E139):

a[0] vai ocupar as posições E100, E101, E102, E103 a[1] vai ocupar as posições E104, E105, E106, E107 a[2] vai ocupar as posições E108, E109, E110, E111 ... ...

a[9] vai ocupar as posições E136, E137, E138, E139

Relembrando... Sessão revisão

Do mesmo modo que dizemos que o endereço da variável x é E2, podemos dizer que o endereço do array a é E100. Isto é, o endereço do array é o endereço do primeiro byte que o array ocupa. De fato, quando declararmos:

```
int a[10];
```

O nome a é o endereço da primeira posição do array. Por outras palavras, a é sinônimo de &a[0] Como tal, é perfeitamente válido fazer coisas deste estilo,

```
int a[10];     int *p;     p = a;     *p = 10;

/* equivalente a dizer a[0] = 10 */
```

Tipos de Dados

1) Tipos de Dados Primitivos

A cada variável está associado um Tipo de Dados. O tipo de dado define quais os valores que a variável pode conter. Se, por exemplo, dissermos que uma variável é do tipo Inteiro, não poderemos lá colocar um valor Real ou um Caractere.

Ao declararmos o tipo de dados de uma variável, estamos a definir, não só, o tipo de valores que esta pode conter, mas também quais as operações que com elas podemos realizar.

Exemplo de tipos primitivos:

- INT
- REAL
- CHAR
- BOOL

Tipos de Dados

2) Tipos de Dados Composta homogêneas

O nome mais comum atribuído as variáveis compostas homogêneas é vetor ou matrizes.

Vetores, matrizes ou *arrays* correspondem a um tipo de dado utilizado para representar (armazenar e manipular) uma coleção de valores do mesmo tipo.

Uma outra definição para vetores ou matrizes corresponde a um conjunto de variáveis, do mesmo tipo, identificadas por um único nome, onde cada variável (posição) é referenciada por meio de número chamado de índice. Os colchetes são utilizados para conter o índice.

Tipos de Dados

2) Tipos de Dados Composta homogêneas

As variáveis compostas homogêneas são estruturas de dados que caracterizam-se por um conjunto de variáveis do mesmo tipo. Elas pode ser unidimensionais ou multidimensionais.

Unidimensionais (*vetores*)

A variável composta homogênea unidimensional caracteriza-se por dados agrupados linearmente numa única direção, como uma linha reta.

Multidimensionais (matrizes)

A variável composta multidimensional caracteriza-se por dados agrupados em diferentes direções, como em um cubo;

Tipos de Dados

3) Tipos de Dados Compostos Heterogêneos:

As estruturas heterogêneas são conjuntos de dados formados por tipos de dados primitivos diferentes (campos do registro) em uma mesma estrutura.

Registros

Estrutura de Dados – STRUCT

As estruturas de dados consistem em criar apenas um dado que contém vários membros, que nada mais são do que outras variáveis. De uma forma mais simples, é como se uma variável tivesse outras variáveis dentro dela. A vantagem em se usar estruturas de dados é que podemos agrupar de forma organizada vários tipos de dados diferentes, por exemplo, dentro de uma estrutura de dados podemos ter juntos tanto um tipo float, um inteiro, um char ou um double.

As variáveis que ficam dentro da estrutura de dados são chamadas de membros.

```
struct nome_da_estrutura { tipo_de_dado nome_do_membro; };
```

Registros

Ponteiro de Struct

Um struct consiste em vários dados agrupados em apenas um. Para acessarmos cada um desses dados, usamos um ponto (.) para indicar que o nome seguinte é o nome do membro.

Um ponteiro guarda o endereço de memória que pode ser acessado diretamente.

Registros

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

struct pessoa
{  int idade;
   char nome[100]; } ;

main()
{
    struct pessoa p[10];
    p[1].idade  = 30;
    strcpy(p[1].nome,”Timoteo");
    p[2].idade  = 21;
    strcpy(p[2].nome,“Joao");
    p[3].idade  = 10;
    strcpy(p[3].nome,“Alberto");
    p[4].idade  = 40;
    strcpy(p[4].nome,“Marcia");
    printf("\n%d  - %s", p[1].idade, p[1].nome);
    printf("\n%d  - %s", p[2].idade, p[2].nome);
    printf("\n%d  - %s", p[3].idade, p[3].nome);
    printf("\n%d  - %s", p[4].idade, p[4].nome);
    system("pause");
 };
```