Tipos de Dados Definidos pelo Usuário

Tipos de Dados Definidos pelo Usuário

1. Agregados Heterogêneos (Estruturas)

Um agregado heterogêneo agrupa várias variáveis numa só. Sendo utilizados para agrupar um conjunto de dados não similares formando um novo tipo de dado.

Tipos de Dados Definidos pelo Usuário

1. Agregados Heterogêneos (continuação)

Para se criar um agregado heterogêneo usa-se o comando **struct**. Sua forma geral é:

```
struct nome_do_tipo_da_estrutura
{
    tipo_1   nome_campo1;
    tipo_2   nome_campo2;
    ...
    tipo_n   nome_campon;
} variáveis_do_tipo_da_estrutura;
```

UNIVASF

Tipos de Dados Definidos pelo Usuário

```
struct nome_do_tipo_da_estrutura
{
    tipo_1    nome_campo1;
    tipo_2    nome_campo2;
    ...
    tipo_n    nome_campon;
};
struct
{
    tipo_1    nome_campo1;
    tipo_2    nome_campo2;
    ...
    tipo_n    nome_campon;
} variáveis_estrutura;
```

Tipos de Dados Definidos pelo Usuário

1. Agregados Heterogêneos (continuação)

Vamos criar um agregado heterogêneo para armazenar um endereço:

```
struct tipo_endereco
{
    char rua [50];
    int numero;
    char bairro [20];
    char cidade [30];
    char sigla_estado [3];
    long int CEP;
};
```

Tipos de Dados Definidos pelo Usuário

1. Agregados Heterogêneos (continuação)

Vamos agora criar uma estrutura chamada ficha_pessoal capaz de armazenar os dados pessoais de uma pessoa:

```
struct ficha_pessoal
{
    char nome [50];
    long int telefone;
    struct  tipo_endereco  endereco;
};
```

Tipos de Dados Definidos pelo Usuário

1. Agregados Heterogêneos (continuação)

Assim como ocorria nos agregados homogêneos (vetores) , nos agregados heterogêneo também se faz necessário acessar individualmente seus elementos.

Para isso será utilizado o operador “.”.

Veremos sua utilização no exemplo a seguir.

UNIVASF

```c
#include <stdio.h>
#include <string.h>
/*aqui entrariam as declarações das struct's
vistas anteriormente na sequência correta*/
int main ()
{
    struct ficha_pessoal ficha;
    strcpy (ficha.nome, "Fulano de Tal");
    ficha.telefone=4921234;
    strcpy (ficha.endereco.rua,"Rua das Flores");
    ficha.endereco.numero=10;
    strcpy (ficha.endereco.bairro,"Cidade Velha");
    strcpy (ficha.endereco.cidade,"Belo Horizonte");
    strcpy (ficha.endereco.sigla_estado,"MG");
    ficha.endereco.CEP=31340230;
}
```

Tipos de Dados Definidos pelo Usuário

2. Definição de tipo

O comando typedef é utilizado para definir um novo tipo de dado. Ele é utilizado da seguinte forma

```
typedef  tipo  nome_do_tipo;
```

Tipos de Dados Definidos pelo Usuário

Exemplo:

```
typedef struct
{
    int dia;
    int hora;
    int minuto;
} data;
```

```
data d;
```

Tipos de Dados Definidos pelo Usuário

Exercício:

Construa um programa que manipule um vetor com 5 registros de alunos, onde cada registro possui informações referentes ao nome, data de nascimento, número de matricula, CPF e coeficiente de rendimento do aluno. A manipulação do vetor deve ser feita através das seguintes funções: inicializar vetor, imprimir um determinado registro com base no valor do campo CPF e imprimir um determinado registro com base em sua posição no vetor. O programa não pode possuir variáveis globais, deve se utilizar de forma satisfatória das funções mencionadas e deve definir um novo tipo de dado.

UNIVASF

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define num_reg 5
typedef struct
{
    char nom[30];
    char dat[9];
    unsigned long int n_mat;
    char cpf[13];
    float cr;
} registro;
void inicializar_vet (registro *);
void imprimir_reg_pos (registro *, unsigned int);
void imprimir_reg_cpf (registro *, char *);
```

```c
int main()
{
    registro v[num_reg];
    int i;
    char cpf[13];
    inicializar_vet(v);
    printf("\nPosicao do registro a ser impresso: ");
    scanf("%d",&i);
    imprimir_reg_pos (v,i);
    printf("\n\n\nCPF do registro a ser impresso: ");
    scanf("%s",cpf);
    imprimir_reg_cpf (v,cpf);
    return 0;
}
```

```c
void inicializar_vet (registro *v)
{
    int i;
    for (i=0; i<num_reg; i++) {
        printf("\nEntre com as informacoes do %d° registro.",i+1);
        printf("\nNome: ");
        scanf("%29[^\n]", v[i].nom);
        printf("\nData (dd/mm/aa): ");
        scanf("%8[^\n]", v[i].dat);
        printf("\nNumero de matricula: ");
        scanf("%ld",&v[i].n_mat);
        printf("\nCPF: ");
        scanf("%12[^\n]", v[i].cpf);
        printf("\nCoeficinte de rendimento: ");
        scanf("%f",&v[i].cr);
    }
}
```

```c
void imprimir_reg_pos (registro *v, unsigned int pos)
{
    if (pos>num_reg || !pos)
    {
        printf("\nErro!\nPosicao invalida para impressao!\n");
        exit(1);
    }
    else
    {
        printf("\n\n\n%d° registro.\n",pos);
        printf("\nNome: \t\t\t\t%s",v[pos-1].nom);
        printf("\nData: \t\t\t\t%s",v[pos-1].dat);
        printf("\nNumero de matricula: \t\t%ld",v[pos-1].n_mat);
        printf("\nCPF: \t\t\t\t%s",v[pos-1].cpf);
        printf("\nCoeficinte de rendimento: \t%.3f",v[pos-1].cr);
    }
}
```

```c
void imprimir_reg_cpf (registro *v, char *chave)
{
    int i;
    for (i=0; i<num_reg; i++)
        if(!strcmp(v[i].cpf,chave))
        {
            printf("\n\n\n%d° registro.\n",i+1);
            printf("\nNome: \t\t\t\t%s",v[i].nom);
            printf("\nData: \t\t\t\t%s",v[i].dat);
            printf("\nNumero de matricula: \t\t%ld",v[i].n_mat);
            printf("\nCPF: \t\t\t\t%s",v[i].cpf);
            printf("\nCoeficinte de rendimento: \t%.3f",v[i].cr);
            return;
        }
    printf("\n\nNao existe registro com o CPF especificado.\n");
}
```

Alocação Dinâmica de Memória
Parte 1 – Funções malloc e calloc

UNIVASF

Alocação Dinâmica de Memória

Existem duas maneiras de um programa em C armazenar dados na memória principal do computador.

A primeira, utilizando variáveis locais e globais. O que exige que o programador saiba, de antemão, a quantidade de armazenamento necessária para todas as situações a que o programa será exposto.

UNIVASF

Alocação Dinâmica de Memória

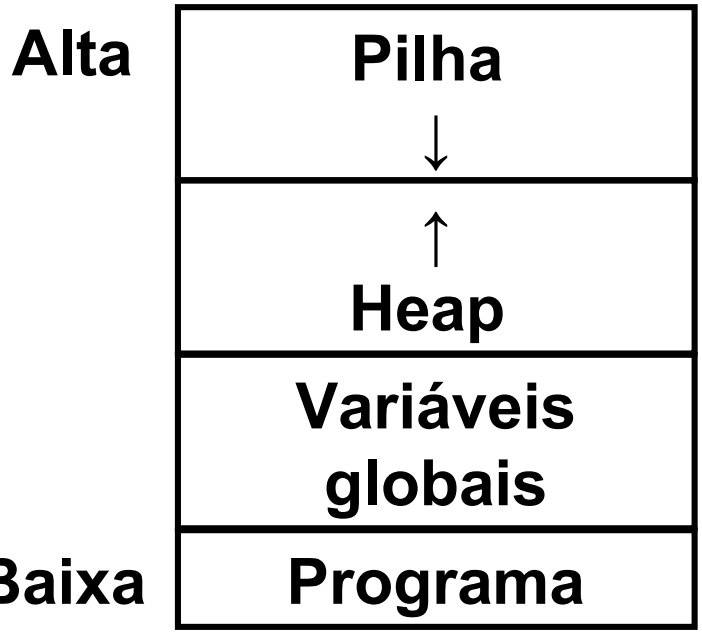
Na segunda, denominada alocação dinâmica de memória, a área para o armazenamento dos dados é alocada na memória livre, também chamada de *heap*.

O *heap* fica situado entre o programa, com sua área de armazenamento permanente, e a pilha. Conforme a imagem a seguir.

UNIVASF

Alocação Dinâmica de Memória

Memória do Sistema



A “Linguagem C” padrão ANSI define apenas 4 funções para o sistema de alocação dinâmica, disponíveis na biblioteca **stdlib.h**.

Alocação Dinâmica de Memória

- malloc

A função **malloc()** serve para alocar memória dinamicamente e tem a seguinte forma:

void *malloc (unsigned int);

```c
#include <stdio.h>
#include <stdlib.h>
main ()
{
    int *p, a, i;
    /* ... */
    p = (int *) malloc (a*sizeof(int));
    if (!p) /*A função retorna NULL se não conseguir alocar a memória solicitada*/
    {
        printf ("** Erro: Memoria Insuficiente **");
        exit (1);
    }
    for (i=0; i<a ; i++)
        p[i] = i*i;
    /* ... */
}
```

Alocação Dinâmica de Memória

Exercício:

Construa um programa que leia da entrada padrão o número de linhas e de colunas de uma matriz de floats, aloque espaço dinamicamente para esta e a inicialize, com valores fornecidos pelo usuário, através da entrada padrão. Ao final o programa deve retornar a matriz na saída padrão com layout apropriado.

UNIVASF

```c
#include <stdio.h>
#include <stdlib.h>
main () {
    int i,j,cont, cont2;
    float *matriz;
    printf ("\nEntre com o numero de linhas da matriz: ");
    scanf("%d",&i);
    printf ("\nEntre com o numero de colunas da matriz: ");
    scanf("%d",&j);
    matriz=(float*)malloc (i*j*sizeof(float));
    if (!matriz) {
        printf ("\nERRO!\n");
        exit (1);
    }
```

```
    for (cont=0;cont<i*j;cont++) {
        printf("\nEntre com o elemento da matriz[%d][%d]: ",
          (cont/j)+1,cont%j+1);
        scanf("%f", matriz+cont);
    }
    for (cont=0;cont<i*j;cont++)
        if (!(cont%j))
            printf ("| %7.2f",matriz[cont]);
        else
            if (cont%j==j-1)
               printf ("%7.2f |\n", matriz[cont]);
            else
               printf (" %7.2f ", matriz[cont]);
}
```

```
    for (cont=0;cont<i;cont++)
        for (cont2=0;cont2<j;cont2++) {
            printf("\nEntre com o elemento da matriz[%d][%d]: ",
              cont+1, cont2+1);
            scanf("%f", matriz+cont*j+cont2);
        }
    for (cont=0;cont<i*j;cont++)
        if (!(cont%j))
            printf ("| %7.2f",matriz[cont]);
        else
            if (cont%j==j-1)
               printf ("%7.2f |\n", matriz[cont]);
            else
               printf (" %7.2f ", matriz[cont]);
}
```

Alocação Dinâmica de Memória

- calloc

A função **calloc()** também serve para alocar memória. Mas, possui uma sintaxe 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 um vetor de **num** elementos de tamanho **size**. Uma grande diferença de **calloc** para **malloc** é que o **calloc zera** todos os bits da **memória alocada**.

UNIVASF

```c
#include <stdio.h>
#include <stdlib.h>
main ()
{
    int *p, a, i;
    /* ... */
    p=(int *)calloc(a,sizeof(int));
    if (!p)
    {
        printf ("** Erro: Memoria Insuficiente **");
        exit (1);
    }
    for (i=0; i<a ; i++)
        p[i] = i*i;
        ...
}
```

Alocação Dinâmica de Memória

Exercício:

Com base no que vimos, construa um programa que aloque dinamicamente memória para um vetor de strings, o números de elementos do vetor e o comprimento máximo das strings pertencentes a este, serão fornecidos pelo usuário, através da entrada padrão. O vetor deve ser inicializado, através da entrada padrão, e posteriormente, impresso na a saída padrão.

UNIVASF