Introdução a Programação

Ponteiros para Estruturas,
Outros Tipos de Estruturas

Tópicos da Aula

- Hoje aprenderemos a trabalhar com ponteiros para estruturas
  - Ponteiros para estruturas
  - Alocação dinâmica de estruturas
  - Passagem por referência de estruturas
  - Vetores de estruturas x Vetores de ponteiros para estruturas
- Aprenderemos também que existe outros formas de tipos estruturados em C
  - Tipo União (Union)
  - Tipos Enumerados (Enum)

Usando Ponteiro para Estruturas

- Podemos ter variáveis do tipo ponteiro para estruturas

```
struct   ponto     {
      float    x ;
      float    y ;
};
int main() {
      struct ponto q;
      struct   ponto*    p ;
      p = &q
    ...
}
```

A variável p armazena o endereço de uma estrutura

Acessando os Membros Através do Ponteiro

- Os membros de uma estrutura são acessados usando seu nome seguido do operador ponto

Podemos acessar os membros do mesmo jeito utilizando ponteiros?

```
struct   ponto    {
      float   x ;
      float   y ;
};
int main() {
      struct ponto q;
      struct   ponto*   p;
      p.x = 7.0;
   ...
}
```

Não !

Não podemos acessar membros de uma estrutura via ponteiro desta forma

Acessando os Membros Através do Ponteiro

- Para acessar os membros de uma estrutura por meio de um ponteiro, existem 2 formas:
  - Usando o operador * seguido da variável dentro de parênteses

```
(*p).x = 7.0;
```

  - Usando o operador ->

```
p->x = 7.0;
```

Precisa de parênteses para variável `p`, senão o compilador entenderia como `*(p.x)` - Errado!

Alocação Dinâmica de Estruturas

- A alocação dinâmica de estruturas é feita de forma similar a como é feito com tipos primitivos

```
struct   pessoa     {
      char  nome[32] ;
      int   idade ;
      double peso;
};

int main() {
    struct pessoa*   p;
    p = (struct pessoa*) malloc(sizeof(struct pessoa));
    strcpy(p->nome,"Ana");
    p->idade = 30;
    ...
}
```

Aloca espaço na memória para uma estrutura que contém um vetor de 32 caracteres (32 bytes), um inteiro (4 bytes) e um double (8 bytes)

Alocação Dinâmica de Estruturas

```
struct   paciente    {
      int*  vetorTemperatura ;
      struct pessoa  individuo;
};
typedef struct paciente Paciente;
int main() {
   Paciente*   p;
   p = (Paciente*) malloc(sizeof(Paciente));
   p->individuo.idade = 30;
   strcpy(p->individuo.nome,"Ana");
   p->vetorTemperatura = (int*) malloc(5 * sizeof(int));
   ...
}
```

Aloca espaço na memória para um vetor com 5 inteiros

Aloca espaço na memória para uma estrutura que contém um ponteiro para inteiro (4 bytes), e uma estrutura pessoa (44 bytes)

Passagem de Estruturas para Funções

```
void   captura  ( struct  ponto   p ) {
  printf( "Digite as coordenadas do ponto (x,y):" ) ;
  scanf ( "%f %f ", &p.x, &p.y )  ;
}
int main() {
      struct ponto p;
      captura(p);
      ...
}
```

Valores da estrutura não podem ser modificados

- Cópia da estrutura na pilha não é eficiente
- É mais conveniente passar apenas o ponteiro da estrutura

Passagem de Ponteiros para Estruturas para Funções

- Uma função para imprimir as coordenadas

```
void   imprime  ( struct  ponto*   p ){
 printf("O ponto fornecido foi:(%f,%f)\n",p->x,p->y);
}
```

- Uma função para ler as coordenadas

```
void   captura  ( struct  ponto   *p ) {
  printf( "Digite as coordenadas do ponto (x,y):" ) ;
  scanf ( "%f %f ", &p->x, &p->y )  ;
}
```

Permite modificar o valor da variável `p`

Passagem de Ponteiros para Estruturas para Funções

```
struct   ponto  {
    float    x ;
    float    y ;
} ;

void   imprime  ( struct  ponto*   p ){
 printf("O ponto fornecido foi:(%f,%f)\n",p->x,p->y);
}
void   captura  ( struct  ponto* p ) {
  printf( "Digite as coordenadas do ponto (x,y):" ) ;
  scanf ( "%f %f ", &p->x, &p->y )  ;
}
int  main (){
   struct   ponto   p ;
   captura(&p);
   imprime(&p);
}
```

Vetores de Estruturas

- Podemos utilizar vetores de estruturas
- Considere o cálculo de um centro geométrico de um conjunto de pontos

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \quad \text{e} \quad \bar{y} = \frac{\sum_{i=1}^{n} y_i}{n}$$

```
struct ponto centrogeometrico(int n,struct ponto* v){
      int   i ;
      struct ponto  p = { 0.0 , 0.0 } ;
      for  ( i = 0 ;   i < n  ;   i++ ){
            p.x  +=  v[i].x ;
            p.y  +=  v[i].y ;
      }
      p.x /=  n ;p.y /=  n ;
      return  p ;
}
```

Endereço inicial do vetor de pontos

Acessando coordenada x do i-ésimo ponto do vetor v

Vetores de Ponteiros para Estruturas

- São úteis quando temos de tratar um conjunto de elementos complexos
  - Exemplo: Vetores de alunos

    Matricula: número inteiro ;
    Nome: cadeia com 80 caracteres ;
    Endereço: cadeia com 120 caracteres ;
    Telefone: cadeia com 20 caracteres ;

  - Em C podemos representar aluno:

```
struct aluno  {
       int  mat ;
       char  nome [81] ;
       char   end [121] ;
       char   tel [ 21 ] ;    } ;
typedef   struct  aluno   ALUNO ;
```

Vetores de Ponteiros para Estruturas

- Se usarmos um vetor com um número máximo de alunos:

```
#define MAX 100
ALUNO tab[MAX];
```

  - Tipo ALUNO ocupa pelo menos 227(=4+81+121+21) bytes
  - Se for usado um número de alunos inferior ao máximo estimado, a declaração de um vetor dessa estrutura representa um desperdício significativo de memória
  - Uma solução é:

```
#define MAX 100
ALUNO* tab[MAX];
```

Usando Vetores de Ponteiros para Estruturas

```
#define   MAX   100
struct   aluno  {
      int   mat ;
      char   nome [81];
      char   end [121] ;
      char   tel [21] ;
  } ;
typedef  struct  aluno   ALUNO;
ALUNO* tab[MAX];

void   inicializa (int n,ALUNO** tab){
    int  i ;
    for(i=0 ;i< n;i++)
      tab [i] = NULL ;
}
```

Todas as posições do vetor de ponteiros guardam endereços nulos

Usando Vetores de Ponteiros para Estruturas

```
void preenche(int n,ALUNO **tab,int i ){
   if ((i < 0)||(i  >= n)){
      printf ("Indice fora do limite do vetor \n");
      return;
   }
   if (tab[i] == NULL ){
      tab[i]= ( ALUNO*) malloc (sizeof(ALUNO));
      printf ("\nEntre com a matricula:");
      scanf ("%d", &tab[i]->mat) ;
      printf ("\nEntre com o nome:") ;
      scanf ("%80[^\n]", tab[i]->nome) ;
      printf ("\nEntre com o endereco:") ;
      scanf ("%120[^\n]", tab[i]->end) ;
      printf ("\nEntre com o telefone:") ;
      scanf ("%20[^\n]", tab[i]->tel) ;
   }
}
```

Espaço alocado para um novo aluno e endereço é armazenado no vetor

Usando Vetores de Ponteiros para Estruturas

```
void retira(int n,ALUNO** tab,int   i ) {
   if ((i < 0)||(i >= n)){
      printf("Indice fora do limite do vetor\n");
      return; /* sai da funcao */
      }
   if (tab[i] != NULL) {
      free(tab[i]);
      tab[i] = NULL;
   }
}
```

Usando Vetores de Ponteiros para Estruturas

```
void imprime(int n,ALUNO** tab,int i ){
   if ((i < 0)||(i >= n)){
      printf ("Indice fora do limite do vetor \n" );
      return ;    /* sai da funcao */
   }
   if (tab[i]!= NULL){
      printf ("Matricula: %d\n",tab[i]->mat);
      printf ("Nome: %s\n",tab[i]->nome);
      printf ("Endereço: %s\n",tab[i]->end);
      printf ("Telefone: %s\n",tab[i]->tel);
   }
}
```

Usando Vetores de Ponteiros para Estruturas

```
void imprime_tudo(int n,ALUNO** tab){
    int i;
    for(i = 0; i < n; i++)
        imprime(n,tab,i);
 }
```

Usando Vetores de Ponteiros para Estruturas

```
int main(){
   ALUNO *tab[10];
   inicializa(10,tab);
   preenche(10,tab,0);
   preenche(10,tab,1);
   preenche(10,tab,2);
   imprime_tudo(10,tab);
   retira(10,tab,0);
   retira(10,tab,1);
   retira(10,tab,2);
   return   0 ;
}
```

Tipo Estruturado `Union`

- Assim como uma **`struct`**, uma **`union`** agrupa um conjunto de tipos de dados (que podem ser distintos) sob um único nome
- **Diferentemente** de uma **`struct`**, uma **`union`** armazena valores heterogêneos em um mesmo espaço de memória
- Apenas um único membro de uma **`union`** pode estar armazenado em um determinado instante
  - A atribuição a um membro da **`union`** sobrescreve o valor anteriormente atribuído a qualquer outro membro

Utiliza-se em casos onde se quer otimizar uso de memória

Definindo uma Union

- Forma Geral :

```
union    nome_do_tipo  {
 declaração de variável 1 ;
 declaração de variável n ;
};
```

- Exemplo:

```
union    numero  {
   char str[32];
   int inteiro ;
   double real;
};
```

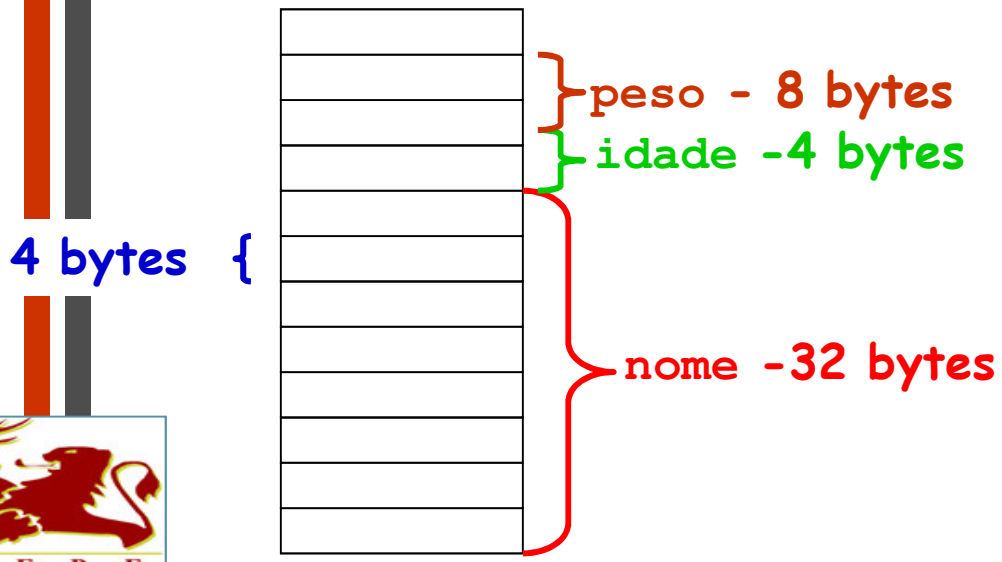
Union x Struct

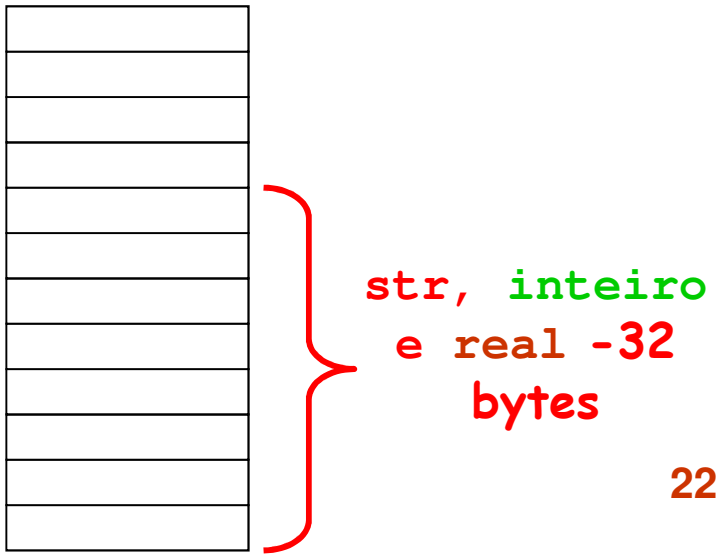
```
struct   pessoa  {
   char nome[32];
   int idade ;
   double peso;
};
```

```
union   numero  {
   char str[32];
   int inteiro ;
   double real;
};
```

44 bytes na memória (soma do tamanho das variáveis)

32 bytes na memória (maior variável - str)





Manipulando uma `Union`

- Declaração de variável

```
union numero num;
```

- Acesso aos membros de uma union
  - Diretamente (Operador “ . ”):

```
num.inteiro = 60;
```

  - Via um ponteiro (Operador “ -> ”):

```
union numero num;
union numero* pnum = &num;
pnum->real = 60.5;
```

Usando Unions

```
union numero {
   char str[32];
   int inteiro ;
   double real;
};
typedef union numero NUMERO;

int main(){
     NUMERO num;
     strcpy(num.str,"Joao");
     num.real = 45.7;
     num.inteiro = 70;
     printf("num.inteiro = %d\n",num.inteiro);
     printf("num.real = %d\n",num.real);
     printf("*num.str = %d\n",*num.str);
     return 0;
}
```

O que será impresso nas 3 linhas da área demarcada?

```
num.inteiro = 70
num.real = 70
*num.str = 70
```

Tipos Estruturado `enum`

- O tipo estruturado enum (enumeração) consiste de um conjunto de constantes inteiras, em que cada uma delas é representada por um nome
- Uma enumeração é uma forma mais elegante de organizar constantes
  - Dá-se um contexto ao conjunto de constantes
  - Uma variável de um tipo enumerado pode assumir qualquer valor listado na enumeração

Definindo uma enum

- Forma Geral :

```
enum   nome_do_tipo  {
 constante1, constante2,constante n
};
```

  - Exemplo:

```
enum   dias_semana  {
  domingo, segunda, terca, quarta,
quinta, sexta, sabado };
```

Valores das Constantes Definidas em uma `enum`

- Internamente, o compilador atribui valores inteiros a cada constante seguindo a ordem em que elas são definidas, começando de 0, depois 1, etc
  - Portanto:

```
enum   dias_semana  {
   domingo, segunda, terca, quarta,
quinta, sexta, sabado };
```

```
domingo == 0
```

```
segunda == 1
```

```
sabado == 6
```

Valores das Constantes Definidas em uma `enum`

- No entanto, o programador pode atribuir diretamente os valores inteiros desejados

```
enum   dias_semana  {
   domingo = 1, segunda, terca, quarta,
quinta, sexta, sabado };
```

```
domingo == 1
```

```
segunda == 2
```

```
sabado == 7
```

Usando enums

```
enum   dias_semana  {
  domingo, segunda, terca, quarta, quinta, sexta,
sabado
};
typedef enum dias_semana DIAS;

int dia_util(DIAS dia) {
    int ehUtil = 0;
    if (dia >= segunda && dia <= sexta) {
        ehUtil = 1;
    }
    return ehUtil;
}
```

Função verifica se um dia passado como argumento é útil ou não