Universidade de Brasília

Departamento de Ciência da Computação

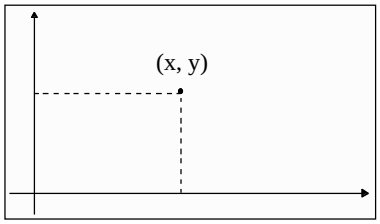
Curso C: Estruturas

Prof. Ricardo Pezzuol Jacobi
rjacobi@cic.unb.br

Estruturas

- *Struct* são coleções de dados heterogêneos agrupados em uma mesma estrutura de dados
- Ex:

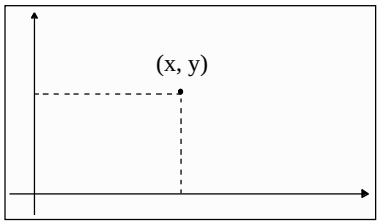
  armazenas as coordenadas (x,y) de um ponto:



Estruturas

- Declaração:

```
struct {
  int x;
  int y;
} p1, p2;
```



- a estrutura contém dois inteiros, *x* e *y*
- *p1* e *p2* são duas variáveis tipo *struct* contendo duas coordenadas cada.

Declaração

- Formato da declaração:

```
struct nome_da_estrutura {
      tipo_1 dado_1;
      tipo_2 dado_2;
      ...
      tipo_n dado_n;
} lista_de_variaveis;
```

- A estrutura pode agrupar um número arbitrário de dados de tipos diferentes
- Pode-se nomear a estrutura para referencia-la

Nomeando uma Estrutura

```
struct {
  int x;
  int y;
} p1;

struct {
  int x;
  int y;
} p2;
```

Repetição 

• *struct ponto* define um *novo tipo de dado*

```
struct ponto {
      int x;
      int y;
};
struct ponto p1, p2;
```

• Pode-se definir novas variáveis do tipo *ponto*

Estruturas

- acesso aos dados:

`struct-var.campo`

- Ex:

```
p1.x = 10;  /*atribuição */
p2.y = 15;
if (p1.x >= p2.x) &&
   (p1.y >= p2.y) ...
```

Atribuição de Estruturas

- Inicialização de uma estrutura:

```
struct ponto p1 = { 220, 110 };
```

- Atribuição entre estruturas *do mesmo tipo*:

```
struct ponto p1 = { 220, 110 };
struct ponto p2;

p2 = p1;  /* p2.x = p1.x e p2.x = p1.y */
```

- Os campos correspondentes das estruturas são automaticamente copiados do destino para a fonte

Atribuição de Estruturas

- Atenção para estruturas que contenham ponteiros:

```
struct aluno {
   char *nome;     int idade;
} a1, a2;

a1.nome = "Afranio";
a1.idade = 32;
a2 = a1;
```

- Agora a1 e a2 apontam para o mesmo *string* nome:

```
a1.nome == a2.nome == "Afranio"
```

Composição de Estruturas

```
struct retangulo {
  struct ponto inicio;
  struct ponto fim;
};
struct retangulo r = { { 10, 20 }, { 30 , 40 } };
```

- Acesso aos dados:

```
r.inicio.x += 10;
r.inicio.y -= 10;
```


Estruturas como parâmetros

```
struct ponto cria_ponto (int x, int y) {
    struct ponto tmp;

    tmp.x = x;
    tmp.y = y;
    return tmp;
}

main () {
    struct ponto p = cria_ponto(10, 20);
}
```

Operações

- operações entre membros das estruturas devem ser feitas membro a membro:

```
/* retorna uma cópia de p1 = p1 + p2 */
struct soma_pts (struct ponto p1, struct ponto p2)
{
 p1.x += p2.x;
 p1.y += p2.y;

 return p1;     /* retorna uma copia de p1 */
}
```


Ponteiros para Estruturas

- estruturas grandes são passadas como parâmetro de forma mais eficiente através de ponteiros

```
struct ponto *pp;
struct ponto p1 = { 10, 20 };
pp = &p1;
printf("Ponto P1: (%d %d)\n", (*pp).x, (*pp).y};
```

- acesso via operador "->":

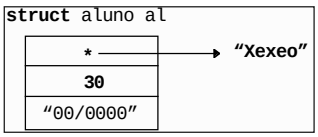
```
printf("Ponto P1: (%d %d)\n", pp->x, pp->y};
```

Arrays de Estruturas

```
struct ponto arp[10];
/* cria um array de 10 pontos */
arp[1].x = 5; /*atribui 5 a coordenada x do 2º ponto */

struct jogador {
  char *nome;
  int idade;
};
struct jogador Brasil[11] = {
      "Felix",    32,
      "Carlos Alberto", 24, ...
};
```


Espaço Alocado para um Estrutura

```
struct aluno {
  char *nome;           /* ponteiro 4 bytes */
  short idade;          /* 2 bytes */
  char matricula[8];    /* array 8 bytes */
};

struct aluno al;
al.nome = "Xexeo";
al.idade = 30;
strcpy(al.matricula, "00/0001");
```



Função *sizeof(tipo)*

- A função *sizeof(tipo)* retorna o tamanho em bytes ocupado em memória pelo tipo de dado passado como parâmetro
- Ex:

```
sizeof(int)             => 4 bytes
sizeof(char)            => 1 byte
sizeof(struct ponto)    => 8 bytes
sizeof(struct ponto *)  => 4 bytes
```


Espaço Efetivo

```
/* teste com Symantec C, PowerPC 603 - Macintosh */

 struct aluno {
   char *nome;              /* 4 bytes */
   short idade;             /* 2 bytes */
   char matricula[3];       /* 3 bytes */
 };
                 /* sizeof(aluno) = 12 bytes */
```

Espaço Efetivo

```
/* teste com Symantec C, PowerPC 603 - Macintosh */


struct aluno1 {
  char *nome;                 /* 4 bytes */
  short idade;                /* 2 bytes */
  char matricula[5];          /* 5 bytes */
};

     /* sizeof(aluno1) = 12 bytes */
```


Espaço Efetivo

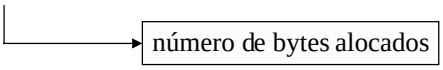
```
/* teste com Symantec C, PowerPC 603 - Macintosh */

struct aluno2 {
  char *nome;           /* 4 bytes */
  short idade;          /* 2 bytes */
  char matricula[7];    /* 7 bytes */
};
           /* sizeof(aluno2) = 16 bytes */
```

- no PowerMac uma estrutura é um múltipo do tamanho da palavra, 32 bits

Alocação Dinâmica de Memória

- as funções *calloc* e *malloc* permitem alocar blocos de memória em tempo de execução

```
void * malloc(size_t n);
```




```
/*
  retorna um ponteiro void para n bytes de
  memória não  iniciados. Se não há memória
  disponível malloc retorna NULL
*/
```


Alocação Dinâmica de Memória

```
void * calloc(size_t n, size_t size);
/*
 calloc retorna um ponteiro para um array com n
 elementos de tamanho size cada um ou NULL se não
 houver memória disponível. Os elementos são
 iniciados em zero
*/
```

- o ponteiro retornado por *malloc* e *calloc* deve ser convertido para o tipo de ponteiro que invoca a função:

Alocação Dinâmica de Memória

```
int *pi = (int *) malloc (sizeof(int));
/* aloca espaço para um inteiro */

int *ai = (int *) calloc (n, sizeof(int));
/* aloca espaço para um array de n inteiros */
```

- toda memória não mais utilizada deve ser liberada através da função *free()*:

```
free(ai);  /* libera todo o array */
free(pi);  /* libera o inteiro alocado */
```


Estrutras Autoreferenciadas

- É possível definir dentro de uma estrutura um ponteiro para a própria estrutura:

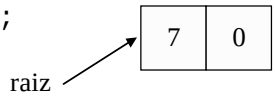
```
struct Qualquer {
  tipo_dado_1 dado_1;
  ...
  struct Qualquer *proximo;
}
```

- *próximo* aponta para uma estrutura do mesmo tipo

Lista de Inteiros

- Exemplo: lista encadeada de inteiros. *raiz* é um ponteiro para o primeiro nodo da lista.

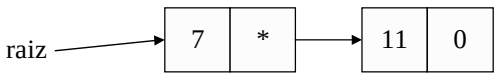
```
struct IntNode {
 int dado;
 struct IntNode *proximo;
} *raiz;

raiz = (IntNode *)malloc(sizeof(IntNode));
raiz->dado = 7;
raiz->proximo = NULL;
```



Lista de Inteiros

- Inserindo elementos na lista:

```
IntNode *pnode;
pnode = (IntNode *)malloc(sizeof(IntNode));
pnode->dado = 11;
pnode->proximo = NULL ; /*terminador de lista*/

raiz->proximo = pnode;
```



Percorrendo a Lista

- Percorre-se uma lista encadeada por meio de um ponteiro:

```
...
IntNode *pnode;
pnode = raiz;
while (pnode != NULL) {
  printf("%d ", pnode->dado);
  pnode = pnode->proximo;
}
```


Insere Nodo

```
struct IntNode *insere_int (int i,
                            struct IntNode *pinicio)
{
struct IntNode *pi;

pi=(struct IntNode *)malloc(sizeof(struct IntNode));
if (pi) {
 pi->dado = i;
 pi->proximo = pinicio;
}
 return pi;
}
```

Print Lista

```
void print_list (struct IntNode *pi) {

 printf("\nLista = ");
 while (pi)  {
  printf (" %d ", pi->dado);
  pi = pi->proximo;
 }
}
```


Lista de Alunos

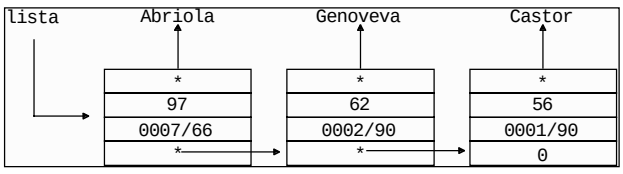
```
struct aluno {

  char *nome;
  short idade;
  char matricula[8];
  struct aluno *prox;
};

struct aluno *ListaAlunos;
/* variavel global */
```

Insere Aluno

```
int insere_aluno (char *n, short id, char *m)
{
  struct aluno *pa;

 pa = (struct aluno *)malloc(sizeof(struct aluno));
 pa->nome = n;
 pa->idade = id;
 strcpy(pa->matricula, m);
 pa->prox = ListaAlunos;
 ListaAlunos = pa;
}
```


Criação Lista Alunos

```
void main () {
 insere_aluno("Castor", 56, "0001/90");
 insere_aluno("Genoveva", 62, "0002/90");
 insere_aluno("Abriola", 97, "0007/66");
}
```



Esvaziando a Lista

```
/* esvazia a lista */
void esvazia_lista () {
struct aluno *tmp;

  while (ListaAlunos != NULL) {
      tmp = ListaAlunos ;
      ListaAlunos = lista->prox;
      free(tmp);
  }
}
```


Impressão Recursiva

```
void imprimeRec ( aluno *lista )
{
 if ( lista != NULL ) {
      printf ("Aluno: %s \n", lista->nome);
      imprimeRec (lista->prox);
      }
}
```