Notas de aula – Algoritmos e Estrutura de Dados 1 (AE42CP) – ponteiros em structs
Prof. Jefferson T. Oliva

Resumo do que foi visto até agora

- Tipos básicos de dados: int, float, char, double, ...

- Estrutura de dados homogêneas: vetores, matrizes e strings

- Estrutura de dados heterogêneas: struct

```
typedef struct {
    tipo1 campo1;
    tipo2 campo2;
    ...
    tipoN campoN;
} nome_registro;
```

Operações com struct: inicialização, acesso aos elementos e atribuição entre structs

Argumento e retorno de função struct

Aninhamento

Vetores de struct

Na aula teórica seguinte, vimos conceitos de ponteiros

- Operador &: endereço de

```
int main(void){
     int i = 19;
     char c = ’z’;
     printf("Valor de i: %d\n", i);
     printf("Endereco de i: %i\n", &i);
     printf("End. de i em hexadecimal: %x\n", &i);
     printf("Valor de c: %c\n", c);
     printf("Endereco de c: %i", &c);
     printf("End. de c em hexadecimal: %x\n", &c);
}
```

- Endereço de: &

- Declaração de ponteiro: tipo *nome;

- Acesso ao conteúdo apontado pelo ponteiro: *nome

- Acesso ao endereço apontado pelo ponteiro: nome

- Endereço do ponteiro: &nome

- Passagem de parâmetros por cópia e referência: func1(int a), func2(int *a)

- Em outras palavras, vimos dois operadores: & e *

```
int main(void){
      int i;
      int *p_i;
      i = 30;
      p_i = &i;
      printf("%d\n", i);
      printf("%d\n", *p_i); // imprime valor no endereço
      printf("%d\n", &i); // imprime valor no endereço de i
      printf("%d\n", p_i);// imprime o endereço armazenado
      return 0;
}
```

- Aritmética de ponteiros

```
char c = ’0’;
int i = 0;
char *pc;
int *pi;
pc = &c;
pi = &i;
pc++; // desloca 1 byte
pi++; // desloca 4 bytes
```

- Comparação de ponteiros

- Vetores

- Ponteiros genéricos: void *p

```
int main(void){
    char c = ’Q’;
    char *pc;
    void *p;

    p = &c;

    printf("Char: %c\n", c);
    printf("ponteiro: %c\n", *(char*) p);
    return 0;
}
```

Aula de hoje: Ponteiros em Structs e Outros Tipos de Estruturas

Ponteiros em Structs

Assim como qualquer tipo de dado, o uso de ponteiros segue a mesma regra para as structs

```
typedef struct aluno{
    char nome[101];
    int RA;
    float coef;
}Aluno;

int main(void){
    Aluno variavel;
    Aluno *ponteiro;
    ponteiro = &variavel;
    return 0;
}
```

Um ponteiro de struct recebe o endereço de uma struct do mesmo tipo.

```
int main(void){
    Aluno variavel;
    Aluno *ponteiro;
    ponteiro = &variavel;

    (*ponteiro).RA = 12345;
    (*ponteiro).coef = 0.65;

    strcpy((*ponteiro).nome, "Gil B. Away");

    return 0;
}
```

Imprimir o endereço do registro: printf("%d\n", ponteiro);

Imprimir um campo do registro: printf("%d\n", (*ponteiro).RA);

Imprimir o endereço de um campo do registro: printf("%d\n", &(*ponteiro).RA);

Cuidado: A expressão *var.campo é equivalente a (*var.campo), mas tem significado muito diferente de (*var).campo. Os dois primeiros não funcionam para o exemplo anterior! Mas para o próximo sim:

- O uso de *var.campo e (*var.campo) são usadas para variáveis de estrutura não declaradas como ponteiros, mas que possuem campos declarados como ponteiros.

- (*var).campo é utilizada quando uma variável de estrutura é declarada como ponteiro

```
typedef struct {
    char nome[101];
    int *RA;
    float coef;
}Aluno2;

int main(void){
   Aluno2 v2;
   Aluno2 *p2;
   int ra = 333;

   v2.RA = &ra;
   printf("%d\n", *v2.RA); // acesso ao campo declarado como ponteiro de uma variável do tipo Aluno2
   *v2.RA = 222;
   printf("%d\n", *v2.RA);

   p2 = &v2;

   printf("%d\n", *(*p2).RA); // acesso ao campo declarado como ponteiro a partir de um ponteiro do tipo Aluno2
   //printf("%d\n", *p2);

   return 0;
}
```

Os ponteiros de struct possuem um operador para abreviar o comando de acesso ao valor: comando -> (comando exclusivo de structs)
- (*p). pode ser substituído por p->

```
int main(void){
    Aluno variavel;
    Aluno *ponteiro;

    ponteiro = &variavel;
    ponteiro->RA = 12345;
    ponteiro->coef = 0.65;
    strcpy(ponteiro->nome, "Roberto");

    return 0;
}
```

No exemplo 2:

```
int main(void){
   Aluno2 v2;
   Aluno2 *p2;
   int ra = 333;

   v2.RA = &ra;
   printf("%d\n", *(*v2).RA); // ou printf("%d\n", *v2->RA));
   *v2.RA = 222;
   printf("%d\n", *v2.RA);

   p2 = &v2;

   printf("%d\n", *p2->RA);

   return 0;
}
```

O uso de ponteiros de struct em funções é como qualquer variáve

```
typedef struct{
   int dia, mes, ano;
}data;

data* diferenca(data *d1, data *d2){
   data aux;
   data *p_aux;

   p_aux = &aux;

   p_aux->dia = d1->dia - d2->dia;
   p_aux->mes = d1->mes - d2->mes;
   p_aux->ano = d1->ano - d2->ano;

   return p_aux;
}

int main(void){
   data d1 = {12, 12, 2019};
   data d2 = {11, 11, 2009};
   data *d3 = diferenca(&d1, &d2);

   printf("%d\n", d3->dia);
   printf("%d\n", d3->mes);
   printf("%d\n", d3->ano);
}
```

Union

Uma união é um formato de dados que pode armazenar tipos diferentes mas apenas um tipo de cada vez

Uma struct pode armazenar um int e um char e um double, por exemplo

Uma união pode armazenar um int ou um long ou um double. Enquanto uma struct armazena um valor para cada registro, uma union armazena um único valor para todos os registros.

```
union uniao{
    tipo1 va1;
    ...
    tipoN varN;
};

typedef uniao Uniao;
```

ou

```
typedef union{
    tipo1 va1;
    ...
    tipoN varN;
}Uniao;
```

Exemplo 1

```
typedef union {
    int val_int;
    long val_long;
    double val_double;
}umpratodos;

int main() {
     umpratodos u;

    u.val_int = 15;
    printf("%d\n", u.val_int);

    u.val_double = 1.38;
    printf("%g\n", u.val_double);
    printf("%d\n", u.val_int);

    return 0;
}
```

Exemplo 2

```
typedef union {
   long nro_id;
   char char_id[20];
}identificador;

typedef struct {
   char marca[20];
   identificador id;
   int tipo;
}inventario;

int main() {
   inventario inv;

   strcpy(inv.marca, "Doli");
   inv.id.nro_id = 5678;

   return 0;
}
```

Ponteiro de união

```
typedef union {
    int val_int;
    long val_long;
    double val_double;
}umpratodos;

int main() {
    umpratodos u;
    umpratodos *p_u;

    p_u = &u;

    u.val_int = 15;
    printf("%d\n", p_u->val_int);
    u.val_double = 1.38;
    printf("%g\n", p_u->val_double);
    printf("%g\n", p_u->val_int);

    return 0;
}
```

Pode ter uma struct dentro de uma união? Resposta: sim.

```
typedef struct{
   int dia, mes, ano;
}data;

typedef union {
    int val_int;
    long val_long;
    double val_double;
   data val_data;
}umpratodos;

int main() {
    umpratodos u;

    u.val_data.dia = 22;
    u.val_data.mes = 8;
    u.val_data.ano = 2019;

    printf("%d\n", u.val_data.ano);
    printf("%d\n", u.val_data.dia);
    printf("%d\n", u.val_data.mes);
    printf("%d\n", u.val_int);
    printf("%d\n", u.val_double);

    return 0;
}
```

União anônima

```
typedef struct {
    char marca[20];
    union{ // formato depende do tipo inventario
        long nro_id; // inventários do tipo 1
        char char_id[20]; // outros inventários
    };
    int tipo;
}inventario;

int main() {
    inventario inv;
    strcpy(inv.marca, "Dolly");
    inv.nro_id = 5678;

    return 0;
}
```

Enumerados

Define um enumerado como um novo tipo

Estabelece símbolos (palavas) como constantes simbólicas para números inteiros entre 0 e o número de símbolos

enum enumerado {simb1, ..., simb1};

enum espectro {vermelho, laranja, amarelo, verde, azul, violeta, anil, ultravioleta};

ou

typedef enum {vermelho, laranja, amarelo, verde, azul, violeta, anil, ultravioleta}espectro;

Exemplos:
espectro banda;

```
banda = azul; // válido, azul é um enumerador
banda = 2000; // inválido!, 2000 não é um enumerador
banda = laranja; // válido
banda++; // inválido
banda = laranja + vermelho; // válido
...
int cor = azul; // válido, tipo espectro promovido a int
banda = 3; // válido, o tipo espectro atribui um valor para cada tipo
cor = 3 + vermelho; // válido, vermelho é convertido para int
...
banda = espectro(3); // erro
```

Estabelecendo valores para enumeradores

```
enum bits {um = 1, dois = 2, quatro = 4, oito = 8};

enum grandepasso { primeiro, segundo = 100, terceiro};

enum {zero, nulo = 0, um, numero_um = 1};
```

Referências

Arakaki, R.; Arakaki, J.; Angerami, P. M.; Aoki, O. L.; Salles, D. S. Fundamentos de programação C: técnicas e aplicações. LTC, 1990.

Deitel, H. M.; Deitel, P. J. Como programar em C. LTC, 1999.

Pereira, S. L. Estrutura de Dados e em C: uma abordagem didática. Saraiva, 2016.

Tenenbaum, A.; Langsam, Y. Estruturas de Dados usando C. Pearson, 1995.