Linguagem C ANSI:
Aula 5 – Tipos de Dados Definidos pelo Usuário

Software Básico, turma A

(Baseado no Curso de Linguagem C da UFMG)

Prof. Marcelo Ladeira – CIC/UnB

Sumário da Aula 5

- Tipos de Dados Definidos pelo Usuário
  - Estruturas
  - Declaração union
  - Campos de Bits
  - Operador sizeof
  - Facilidade typedef
  - Estruturas Auto Referenciadas

Estruturas

- Coleção de variáveis agrupadas juntas sob um único nome para facilitar a manipulação
  - Podem ser de tipos diferentes
    - São vetores heterogêneos.
  - Podem ser atribuídas via comando de atribuição e passadas para ou retornadas de funções.
    - Não podem ser comparadas.
    - Estruturas automáticas podem ser iniciadas.
  - Define um novo tipo de dados em C
    - É a única forma de definir novos tipos de dados em C

Estruturas

- **Sintaxe Geral**

```
struct tag_da_estrutura
{
tipo_1 nome_1;
tipo_2 nome_2;
...
tipo_n nome_n;
} variáveis_estrutura;
```

- **Podem ser aninhadas**

- **Tag**
  - **Nome do tipo criado**
    - **Opcional: tipo anônimo. Variáveis_estrutura deve existir.**
  - **Abreviatura para a parte entre chaves.**
- **Membros**
  - **São os componentes.**
    - **Escopo é a estrutura.**
- **Variáveis**
  - **Objetos de dados do novo tipo criado.**
  - **Opcional: tag deve existir.**

Estruturas
Atribuições

- **Somente se forem do mesmo tipo**
  - **Os membros são copiados um a um, de uma estrutura para a outra.**
    - **Todos os campos são copiados.**
      - **Qual a forma eficiente de implementar essa operação?**
    - **Pode provocar efeitos indesejáveis se algum membro for um ponteiro.**
      - **Por quê?**
  - **Não existe operação análoga ao comando Cobol**

    MOVE cadastro-1 TO cadastro-2, BY CORRESPONDING.

Cadastro-1 e cadastro-2 são estruturas com alguns campos em comum (mesmo nome, nível e tipo).

Estruturas
Exemplo de Uso de Estruturas

- **Comentários**
  - **Forma de acesso de membro em expressão**

    variável_estrutura.membro
  - **Podem ser iniciadas**
    - **Informe lista de valores entre chaves.**
  - **Exemplo KR, p. 105 a 107.**

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

```
struct ponto pt;
struct pt = {320, 200};
double dist, sqrt (double);
dist = sqrt((double) pt.x * pt.x +
     (double) pt.y * pt.y);
struct rect {
     struct ponto pt1;
     struct ponto pt2;
};
struct rect screen;
printf ("x=%d", screen.pt1.x);
```

Estruturas
Exemplo de Uso de Estruturas

```
#include <stdio.h>
#include <string.h>
struct tipo_endereco {
      char rua [50];
      int numero;
      char bairro [20];
      char cidade [30];
      char sigla_estado [3];
      long int CEP;
};

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

```
main (void)
{
struct ficha_pessoal ficha;
strcpy (ficha.nome,"Luiz Osvaldo Silva");
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;
return 0;
}
```


UFMG, pág. 116 e 117.

Estruturas
Passando Estruturas para Funções

- Existem três abordagens
  - Passar os componentes separadamente
    - Os membros passam a ser parâmetros da função.
  - Passar a estrutura como um todo
    - O nome da estrutura é passado como parâmetro
      - Os membros são copiados no escopo da função pois a passagem de parâmetros é por valor.
        - Aplicável se a estrutura é pequena.
        - Alterações são locais à função.
  - Passar um ponteiro para a estrutura
    - O ponteiro é passado como parâmetro
      - Passagem por referência. Não há cópia da estrutura.
        - Indicada quando a estrutura for grande.

Estruturas
Passando Estruturas para Funções

```
struct ponto {
      int x;
      int y;
};
struct retang {
      struct ponto pt1;
      struct ponto pt2;
};
struct ponto c_ponto (int x, int y)
{
      struct ponto temp;
      temp.x = x;
      temp.y = y;
      return temp;
}
struct retang janela;
struct ponto meio;
```

- **Passando apenas os membros**

```
janela.pt2 = c_ponto(XMAX,
      YMAX);
meio = c_ponto ( (janela.pt1.x +
      janela.pt2.x)/2, (janela.pt1.y +
      janela.pt2.y)/2 );
```

- **Passando toda a estrutura**

```
struct ponto s_ponto (struct ponto
      p1, struct ponto p2)
{
      p1.x += p2.x;
      p1.y += p2.y;
      return p1;
}
```

KR, pág. 107

Estruturas
Passando Estruturas para Funções

- Passando ponteiro para a estrutura

```
struct ponto origem, *pp;
pp = &origem;
printf("origem eh (%d,
    %d)\n", (*pp).x, (*pp).y);
```

- **Comentários**
  - **Parênteses necessários**
    - **Prioridade do ponto é maior do que indireção**
      - ***pp.x gera erro porque x não é um ponteiro.**

- **Forma abreviada**

pp->x equivale a (*pp).x

```
printf("origem eh (%d,%d)\n",
    pp->x, pp->y);
```

  - **Associação de . e ->**
    - **da esquerda para direita**
      - **As expressões abaixo são equivalentes**

```
struct retang r, *rp = &r;
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```


KR, pág. 108

Estruturas
Passando Estruturas para Funções

- **Operadores estrutura, *ponto* e ->, chamada de função, ( ), e subscrito, [ ], possuem a mais alta prioridade.**
  - **induzem parênteses**
    - **Exemplo**

```
struct {
   int len;
   char *str;
} q,r,*p=&q;
```

- ++p->len
  - incrementa len

    Parênteses: ++(p->len).
- (++p)->len
  - incrementa p antes de acessar len
- (p++)->len
  - incrementa p depois de acessar len
    - Esses parênteses não são necessários.

KR, pág. 109

Estruturas
Passando Estruturas para Funções

- **Indução de parênteses**
  - **Exemplo**

```
struct {
    int len;
    char *str;
} r,q,*p=&r;
```

*p->str
  - retorna caracter apontado por str
    - parênteses: *(p->str)

*p->str++
  - retorna caracter apontado por str e incrementa o ponteiro str
    - parênteses: (*p->str), ++str

(*p->str)++
  - incrementa o código do caracter apontado por str

*p++->str
  - retorna o caracter apontado por str e incrementa o ponteiro p
    - parênteses: (*p->str), ++p

KR, pág. 109

Estruturas
Matrizes de Estruturas

- **Exemplo**

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

struct ficha {
        char nome [50];
        long int telefone;
        struct t_endereco endereco;
};
```

  - **Acesso a um elementoda matriz**

```
struct ficha servidor [100] ;
servidor[12].endereco.sigla_estado[1]
```

Qual objeto está sendo acessado?;

- **Alocação estática de ponteiros**

```
struct key {
        char *word;
        int count;
} keytab[] = {
        "auto", 0,
        "break", 0,
        "case", 0,
        "char", 0,
        /* ... */
        {"void", 0},
        {"volatile", 0},
        {"while", 0}
};
```

  - **Qual o tamanho da matriz keytab?**

```
#define NKEYS (sizeof keytab /    \
    sizeof(struct key))
```

  - **Posso usar essa expressão em #if?**

    NÃO! Por quê?

Estou alocando ponteiros e os objetos apontados por eles? Se sim, posso declarar ponteiros para float e alocar os floats de forma análoga? NÃO! Então o que está acontecendo?

Declaração union

- **Variável que armazena (em diferentes tempos) objetos de tipo e tamanho diversos**
  - **Compilador gerencia tamanho e alinhamento.**
    - **Aloca espaço para atender ao maior requisito.**
  - **Permite manipular diferentes tipos de dados na mesma área de memória**
    - **Nenhuma informação dependente de máquina é embutida no programa.**
    - **Análoga a estrutura com todos os membros sobrepostos.**
  - **Similar aos registros variantes de Pascal**
    - **O programador é responsável por controlar o tipo correntemente armazenado na** union**.**

Declaração union

- **Exemplo, KR, pág. 121**

```
struct {
    char *nome;
    int flags;
    int u_tipo;
    union {
      int i_val;
      float f_val;
      char *s_val;
    } u;
} tab [NSYM];
```

- **Referência**

```
if (u_tipo == INT)
   printf("%d\n", tab[i].u.i_val);
else if (u_tipo == FLOAT)
   printf("%f\n", tab[i].u.f_val);
else if (u_tipo == STRING)
   printf("%s\n", tab[i].u.s_val);
else
   printf("tipo errado %d em
    u_tipo\n", u_tipo);
```


Qual o objeto associado a tab[i].u.i_val ? E a *tab[i].u.s_val ? Uniões podem ser iniciadas?

Campos de Bits

- Permitem compactar diversos objetos em uma única palavra de máquina
  - Em geral é usado para representar conjuntos de flags formados por alguns bits.
- É um conjunto de bits adjacentes dentro de uma única palavra.
  - O tamanho do campo de bits é definido por uma expressão constante que segue a declaração do membro da estrutura.

Campos de Bits
Exemplo Definição de Flags

- **Exemplo convencional**
  - **Alternativa 1**

```
#define KEYWORD 1
#define EXTERN 2
#define STATIC 4
```

  - **Alternativa 2**

```
enum { KEYWORD = 1,
    EXTERN = 2, STATIC = 4 };
        /* potencia de 2 */
```

  - **Utilização**

```
flags |= EXTERN | STATIC;
flags &= ~(EXTERN | STATIC);
if ((flags & (EXTERN |
    STATIC)) == 0) ...
```

- **Solução campo de bits**

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern : 1;
    unsigned int is_static : 1;
} flags;
```

  - **Utilização**

```
flags.is_extern=flags.is_static=1;
flags.is_extern=flags.is_static=0;

if (flags.is_extern == 0 &&
    flags.is_static == 0) ...
```


Flags de um símbolo em uma tabela de símbolos. KR, página 122.

Campos de Bits

- Comentários
  - Tornam os programas não portáveis.
    - São dependentes da implementação e do hardware.
      - Em algumas um campo pode ultrapassar a fronteira de uma palavra. Em outras não podem.
    - São atribuídos da esquerda para a direita em algumas máquinas e da direita para a esquerda em outras.
      - É mais comum os campos iniciam a partir do bit 0.
  - Não possuem endereços
    - Não podem formar matrizes.
    - O operador & não pode ser aplicado a eles.

Campos de Bits

- **Comentários**
  - **Podem ser declarados apenas para int**
    - **Especifique signed ou unsigned explicitamente.**
  - **Campo sem nome**
    - **Usado apenas para preencher espaço (filler)**

```
struct {
    unsigned int is_keyword     : 1;
    unsigned int is_extern      : 1;
    unsigned int is_static                  : 1;
    unsigned int            : 4;
    unsigned int is_bit_7   : 1
} flags;
```

  - **Largura zero**
    - **Força alinhamento com nova palavra**

Operador sizeof

- Retorna o tamanho (em bytes) de objeto ou tipo de dados
  - Objeto
    - variável, matriz, estrutura, union, enumeração
- Sintaxe Geral

  sizeof nome_do_objeto

  sizeof (nome_do_tipo)
- Comentários
  - Auxilia a garantir portabilidade.
  - Permite calcular o tamanho de tipos definidos pelo usuário.

Qual a relação com o problema de fronteiras endereçáveis?

Operador sizeof

- **Exemplo**

```
#include <stdio.h>
struct t_endereco {
    char rua [50]; int numero; char bairro [20];
    char cidade [30]; char sigla_estado [3]; long int CEP;
};
struct ficha {
    char nome [50]; long int telefone; struct t_endereco endereco;
};
void main (void) {
    struct ficha_pessoal *ex;
    ex = (struct ficha_pessoal *) malloc(sizeof (struct ficha));
    ...
    free(ex);
}
```


UFMG, pág. 124.

Facilidade typedef

- **Cria nome para um tipo de dados existente.**
  - **Não cria novos tipos de dados.**
    - **Não agrega nenhum novo significado (propriedade) aos objetos declarados com o novo nome do tipo de dados.**
  - **Análogo ao #define mas em tempo de compilação**
- **Sintaxe Geral**

  typedef antigo_nome novo_nome;

  Em geral, o nome novo é escrito em maiúsculas.

Facilidade typedef
Motivação Para o Uso

- **Melhorar a legibilidade**

```
typedef char *STRING;
STRING p, lineptr[MAXLINES], alloc (int);
int strcmp (STRING, STRING);
p = (STRING) malloc(100);

typedef struct tnodo *TREE_PTR;
typedef struct tnodo {     /* nodo da árvore binária: */
    char *word;            /* ponteiro para o texto */
    int contador;          /* número de ocorrências */
    struct tnodo *esq;                /* ponteiro para subárvore da esquerda */
    struct tnodo *dir;     /* ponteiro para subárvore da direita */
} TREE_NODO;
TREE_PTR talloc (void)
{
return (TREE_PTR) malloc (sizeof (TREE_NODO));
}
```

Note que com o novo nome para o tipo struct tnodo não é necessário usar o prefixo struct na declaração de novas variáveis desse tipo.

Facilidade typedef
Motivação Para o Uso

- **Melhorar a legibilidade**

  typedef **int** (*PFI)(**char** *, **char** *);

  PFI strcmp, numcmp; /* não funciona com #define */

- **Parametrizar um programa contra problemas de portabilidade**
  - Use typedef para tipos de dados que são dependentes da máquina
    - Somente os typedef precisarão ser alterados
      - Typedefs para **short**, **int** e **long** permitem selecionar o que se deseja adequar ao se migrar para nova máquina e se querer continuar com o mesmo tamanho de objetos.
      - size_t e FILE são typedefs.

Estruturas Auto Referenciadas

- **A estrutura contém um membro com um ponteiro para uma estrutura do seu tipo**
  - **Se o membro for a estrutura e não um ponteiro, a definição fica recursiva e de tamanho infinito.**

```
typedef struct tnodo {          /* nodo da árvore binária: */
    char *word;                 /* ponteiro para o texto */
    int contador;               /* número de ocorrências */
    struct tnodo *esq;          /* ponteiro para filho da esquerda */
    struct tnodo *dir;          /* ponteiro para filho da direita */
} TREE_NODO
```

Estruturas Auto Referenciadas

```
#include <stdio.h>
#include <stdlib.h>
typedef struct tipo_produto {
        int codigo;                     /* Código do produto */
        double preco;                   /* Preço do produto */
        struct tipo_produto *proximo; /* Próximo produto */
} TProduto;
void inserir(TProduto **cabeca);
void listar (TProduto *cabeca);
int main()
{
        TProduto *cabeca = NULL;   /* Ponteiro para a cabeça da
        lista */
        TProduto *noatual;          /* Ponteiro para percorrer a lista */
        char q;    /* Opção do usuário */
        do {
        printf("\n\nOpcoes: \nI -> para inserir novo produto;\nL ->
        para listar os produtos; \nS -> para sair \n:");
        scanf("%c", &q);            /* Lê a opção do usuário */
        switch (q) {
            case 'i': case 'I': inserir(&cabeca); break;
            case 'l': case 'L': listar(cabeca); break;
            case 's': case 'S': break;
            default: printf("\n\n Opção não valida");
        }
        fflush(stdin);                  /* Limpa o buffer de entrada */
        } while ((q != 's') && (q != 'S') );
        /* Desaloca a memória alocada */
        noatual = cabeca;
        while (noatual != NULL) {
        cabeca = noatual->proximo;
        free(noatual);
        noatual = cabeca;
        }
}

        /* Lista os elementos na lista */
void listar (TProduto *noatual)
{
        int i=0;
        while (noatual != NULL) /* Enquanto não fim */
        {
        i++;
        printf("\n\nProduto numero %d\nCodigo: %d
        \nPreco: R$ %.2lf", i, noatual->codigo, noatual-
        >preco);
        noatual = noatual->proximo;
         /* Aponta próximo */
        }
}
```


UFMG, pág. 126 a 128.

Estruturas Auto Referenciadas

```
                /* Lista os elementos na lista */
void listar (TProduto *noatual)
{
        int i=0;
        while (noatual != NULL) { /* Enquanto nao fim */
                i++;
                printf("\n\nProduto número %d\nCódigo:
        %d \nPreço: R$ %.2lf", i, noatual->codigo,
        noatual->preco);
                noatual = noatual->proximo;
                /* Aponta próximo */
        }
}
                /* Função para inserir no fim da lista */
void inserir (TProduto **cabeca) {
        TProduto *noatual, *novono;
        int cod;
        double preco;
        printf("\n Código do novo produto: ");
        scanf("%d", &cod);
        printf("\n Preço do produto:R$ ");
        scanf("%lf", &preco);
        if (*cabeca == NULL) { /* Se lista vazia */

                /* cria o no cabeca */
                *cabeca = (TProduto *) malloc(sizeof(TProduto));
                (*cabeca)->codigo = cod;
                (*cabeca)->preco = preco;
                (*cabeca)->proximo = NULL;
        else  {  /* Se não vazia, insere no fim */
                noatual = *cabeca;
                while (noatual->proximo != NULL)
                                noatual = noatual->proximo;
                /* No fim noatual aponta ultimo no*/
                /* Aloca memoria para o novo no */
                novono = (TProduto *) malloc (sizeof(TProduto));
                novono->codigo = cod;
                novono->preco = preco;
                novono->proximo = NULL;
                /* Faz o ultimo no apontar o novo no */
                noatual->proximo = novono;
        }
}
```