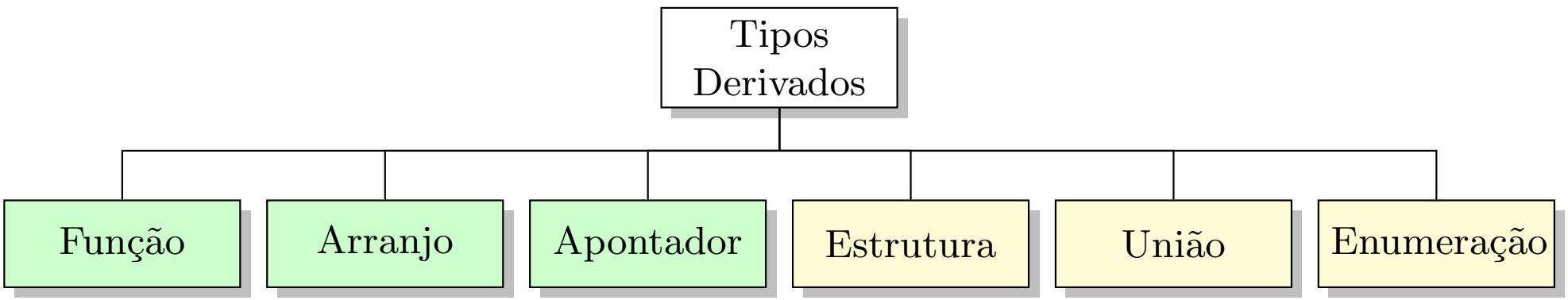
Tipos enumerados, estrutura e união

Alexsandro Santos Soares
prof.asoares@gmail.com

Universidade Federal de Uberlândia
Faculdade de Computação

Introdução

- Já discutimos três dos seis tipos derivados: funções, arranjos e apontadores.
- Nesta aula veremos os três tipos restantes: enumerado, estrutura e união.
- Também discutiremos um construção útil, a definição de tipos sinônimos.



Definição de tipos sinônimos

- Em C, há uma forma de declarar tipos sinônimos que funciona para qualquer tipo.
- Uma definição de tipo sinônimo, usando `typedef`, dá um apelido a tipo de dados que pode ser usando em qualquer lugar onde um tipo for permitido.
- O formato é

```
typedef tipo APELIDO;
```

- Como exemplo, podemos criar um apelido `INTEIRO` para o tipo `int` usando a declaração abaixo:

```
typedef int INTEIRO;
```

- Embora o apelido possa ser definido como qualquer outro identificador em C, sugerimos que ele sempre venha em letras maiúsculas.
  - O mesmo conselho usado para constantes definidas com `#define`.
  - Isso torna o programa mais fácil de ler.

Outro exemplo de uso de typedef

- Um dos usos mais comum da definição de tipos sinônimos é nas declarações complexas.
- Para mostrar o conceito, vamos declarar um arranjo de apontadores para strings:
  - Primeiro sem usar tipo sinônimo:

```
char *arrApontString[20];
```

  - Para simplificar a declaração, usaremos um tipo sinônimo para criar o tipo string e depois definiremos o arranjo usando esse novo tipo:

```
typedef char* STRING;
STRING arrApontString[20];
```

Tipos enumerados

- O tipo enumerado é um tipo definido pelo usuário com base no tipo inteiro.
- O formato para declarar um tipo enumerado é

```
enum nomeDoTipo { constante1, constante2, ..., constanteN};
```

- A cada constante da enumeração é atribuída um valor inteiro.
  - Se não atribuirmos explicitamente os valores, o compilador atribui à primeira constante o valor 0, à segunda o valor 1, à terceira o valor 2 e assim por diante.
- Como exemplo, considere definir um tipo enumerado para cores:

```
enum cor {VERMELHO, AZUL, VERDE, BRANCO};
```

  - O tipo `cor` possui quatro e somente quatro valores possíveis.
  - O intervalo de valores vai de 0 a 3, com `VERMELHO` associado a 0, `AZUL` a 1, `VERDE` a 2 e `BRANCO` a 3.

Tipos enumerados

- Uma vez declarado, um tipo enumerado pode ser usado para criar varáveis como se fosse qualquer outro tipo:

```
enum cor x;
enum cor y;
enum cor z;
```

- Essas variáveis, ou as contantes enumeradas, podem ser usadas em qualquer contexto onde um inteiro puder ser usado.
- Podemos por exemplo atribuir valores a elas:

```
x = AZUL;
y = BRANCO;
z = VERDE;

x = y;
z = y;
```

Comparando tipos enumerados

- Podemos comparar valores enumerados da mesma forma que qualquer outro inteiro:

```
if (cor1 == cor2)
   ...
if (cor1 == AZUL)
   ...
```

- Um uso comum acontece com o comando `switch`:

```
enum mes {JAN, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET, OUT,
          NOV, DEZ};

enum mes mesAtual;

switch( mesAtual ){
  case JAN: ...
            break;
  case FEV: ...
            break;
  ...
} // switch
```

Conversão do tipo enumeração

- Os tipos enumerados podem ser convertidos explicitamente ou implicitamente.
- O compilador pode converter implicitamente um tipo enumerado em um inteiro.
- Entretanto, se tentarmos converte um inteiro em um tipo enumerado podemos obter um aviso ou um erro de compilação.
- Como exemplo, considere

```
enum cor {VERMELHO, AZUL, VERDE, BRANCO};

int x;
enum cor y;

x = AZUL;          // Válido e x conterá 1.
y = 2;             // O compilador reclamará disso.

y = (enum cor) 2; // Válido, pois há uma conversão explícita.
```

Inicializando constantes enumeradas

- Nos slides anteriores deixamos que o compilador decidisse os valores que seriam associados às constantes enumeradas.
- Por exemplo, quando definimos os meses do ano, janeiro recebia o valor 0.
- Assuma agora que janeiro deveria ser o mês 1 e não 0, fevereiro, mês 2 e assim em diante.
- Podemos fazer isso em C da seguinte forma

```
enum mes {JAN = 1, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET, OUT,
                    NOV, DEZ};
```

- Se atribuirmos um valor à uma constante e não às outras, o compilador atribuirá à próxima constante o valor da anterior acrescido de 1.
- Podemos também atribuir valores duplicados às constantes:

```
enum minhacor {VERMELHO, ROSA = 0, JASMIM = 0, ESCARLATE = 0,
               AZUL, AMARELO = 1, VERDE_AGUA = 1,
               VERDE, JADE = 2, BRANCO};
```

Enumerações anônimas: constantes

- Se criarmos um tipo enumerado sem um nome, ele será um tipo enumerado anônimo.
- Devido aos identificadores em tipos enumerados serem constantes, os tipos enumerados são uma forma conveniente de declarar constantes.
- Como exemplo podemos associar nomes a caracteres de pontuação:

```
enum {espaco = ' ', virgula = ',', doispontos = ':'};
```

- Um outro exemplo é

```
enum {FALSO, VERDADEIRO};
```

- Uma enumeração anônima não pode ser usada para declarar uma variável.
- Como `FALSO` vem primeiro ele recebe o valor 0 e poderia ser usando onde precisássemos de um `false`.

Exemplo 1 (Ajustando o efeito de fontes)

Suponha que você esteja projetando um botão para uma aplicação com janelas. Imagine que o usário tenhas as seguintes opções para o efeito no texto que vai sobre esse botão:

- itálico
- negrito
- sublinhado

Note que o usário não precisa selecionar somente uma opção por vez. Ele pode selecionar as três se quiser ou qualquer outra combinação. Vamos escrever um programa que permita ao usuário fazer essa seleção.

```c
#include <stdio.h>

int main(void){
  enum efeitos { NEGRITO = 1,
     ITALICO = 2,
     SUBLINHADO = 4};
  int meusEfeitos = 0;

  meusEfeitos = NEGRITO | SUBLINHADO;

  //    00000001
  //  | 00000100
  //  ------------
  //    00000101

  printf("%d\n\n", meusEfeitos);
  if (meusEfeitos & NEGRITO)
    printf("Negrito\n");

  if (meusEfeitos & ITALICO)
    printf("Itálico\n");
  else
    printf("Não há itálico.\n");

  if (meusEfeitos & SUBLINHADO)
    printf("Sublinhado\n");
  return 0;
} // main
```

Observações sobre o exemplo

- Note no programa que as contantes são potências de 2.
- A razão para isso é que podemos combinar dois ou mais efeitos de uma vez usando o operador `|`, que realiza o ou bit a bit.

Saída da execução do programa:

```
> ./exemplo1.exe

5

Negrito
Não há itálico.
Sublinhado
```

Estrutura

- Uma estrutura é uma coleção de elementos relacionados, possivelmente de diferentes tipos, que possui um único nome.
- Como exemplo, imagine que se deseje guardar informações sobre uma pessoa: seu nome, cpf e salário.
  - Poderíamos ter variáveis diferentes de nomes `nome`, `cpf` e `salario` para guardar essas informações separadamente.
  - Mas, se desejarmos guardar informações sobre 7000 pessoas a abordagem acima é inviável, pois precisaríamos de 3 vezes 7000 variáveis para guardar toda informação.
- Uma abordagem melhor é ter uma coleção de informações relacionadas sob um único nome `Pessoa` e usá-la para qualquer pessoa.
  - O código ficará mais limpo, legível e eficiente.
- Essa coleção de informações relacionadas possuidora de um único nome `Pessoa` é uma estrutura.

Estrutura

- Cada elemento em uma estrutura é chamado de campo.
- Um campo possui todas as características das variáveis que vemos em nossos programas:
  - Ele possui um tipo e um endereço de memória.
  - A ele podem ser atribuídos valores.
  - Podemos acessar um campo para inspeção ou manipulação.
- A única diferença de um campo em relação a uma variável é que um campo faz parte de uma estrutura.
- Já vimos um tipo que pode guardar múltiplas partes de um dado, o arranjo.
  - A diferença entre um arranjo e uma estrutura é que todos os elementos do arranjo devem possuir o mesmo tipo, mas numa estrutura os elementos podem ser de tipos *diferentes*.

Declaração do tipo estrutura

- Em C, há dois modos de se declarar uma estrutura: estruturas rotuladas e estruturas definidas como tipos.
- Uma estrutura rotulada pode ser usada para definir variáveis, parâmetros e tipos de retorno. O formato de declaração é

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

- Como exemplo, a estrutura `Pessoa` poderia ser definida assim:

```
struct Pessoa
{
  char  nome[50];
  int   cpf;
  float salario;
} ;
```

Declaração do tipo estrutura com typedef

- Outra forma de declarar uma estrutura é usar uma definição de tipo com `typedef`, cujo formato é

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

- Podemos então definir um tipo novo `Pessoa` como uma estrutura usando

```
typedef struct
{
  char  nome[50];
  int   cpf;
  float salario;
} Pessoa ;
```

Declaração de variável estrutura

- Após uma estrutura ter sido definida, criamos um tipo definido pelo usuário, mas nenhuma memória será alocada a ela.
- Podemos usar o novo tipo estrutura para declarar variáveis.
- Como exemplo podemos criar variáveis do tipo `Pessoa` de duas maneiras:

Usando estrutura rotulada.

```
struct Pessoa
{
  char  nome[50];
  int   cpf;
  float salario;
} ;

struct Pessoa umaPessoa;
struct Pessoa outraPessoa;
struct Pessoa umaMultidao[7000];
```

Usando `typedef`

```
typedef struct
{
  char  nome[50];
  int   cpf;
  float salario;
} Pessoa;

Pessoa umaPessoa;
Pessoa outraPessoa;
Pessoa umaMultidao[7000];
```

Inicialização

- As regras para inicialização de uma estrutura são similares à aquelas para arranjos.
- Os inicializadores são envoltos em chaves e separados por vírgula.
- Cada um deles deve casar com o tipo correspondente da definição da estrutura.
- Abaixo está o primeiro exemplo de inicialização de estrutura:

```
typedef struct {
  int x;
  int y;
  float t;
  char u;
} Exemplo;

Exemplo ex1 = {2, 5, 3.2, 'A'};
```

- Aqui temos um unicializador para cada campo: o campo `x` recebe 2; `y` recebe 5; `t` recebe 3.2; e o campo `u` recebe `'A'`.

Inicialização

- Usando a mesma definição de estrutura do exemplo anterior, podemos declarar e inicializar uma nova variável como:

```
Exemplo ex2 = { 7, 3 };
```

- Note que nem todos os campos foram inicializados.
- Como nos arranjos, sempre que um ou mais inicializadores estiverem ausentes, os elementos da estrutura receberão valores nulos: zero para inteiros e números em ponto-flutante; o caractere nulo `'\0'` para caracteres e strings.
- Assim, os campos de `ex2` ficarão com os seguintes valores: `x` com 7, `y` com 3, `t` com 0.0 e `u` com `'\0'`.

Acessando os campos da estrutura

- Agora que já sabemos declarar e inicializar estruturas, veremos como usá-las nos programas.
- Primeiro veremos como acessar campos individuais de uma estrutura e depois examinaremos como atribuir toda uma estrutura a outra.
- Em seguida veremos como apontadores podem ser usados com estruturas e concluiremos com um programa que usa estruturas.

Referenciando campos individuais

- Cada campo de uma estrutura pode ser acessado e manipulado usando expressões e operadores.
  - Tudo que podemos fazer com uma variável, podemos fazer com um campo.
- O único problema é identificar o campo individual que estamos interessados.
- Para acessar qualquer campo em uma estrutura usamos o formato

```
nome_da_variável_estrutura.nome_do_campo
```

- Se quisermos acessar o campo `salario` da variável estrutura `umaPessoa`, definida anteriormente, usamos

```
umaPessoa.salario
```

- Podemos então usar essa forma para comparar o salário de uma pessoa com a de outra, para depois dar um aumento à primeira pessoa:

```
if (umaPessoa.salario < outraPessoa.salario)
   umaPessoa.salario += 500.0;
```

Referenciando campos individuais

- Podemos também ler dados para ou escrever dados de campos de estruturas da mesma forma como fazemos com variáveis:

```
scanf("%d %d %f %c", &ex1.x, &ex1.y, &exe1.t, &ex1.u);
```

- Note que o operador de endereço está no início da variável estrutura.

Exemplo 2 (Somando distâncias)

Escrever um programa para somar duas distâncias digitadas pelo usuário. A medida de distância poderá estar em metros e centímetros.

```c
#include <stdio.h>

typedef struct
{
  int metro;
  int centimetro;
} Medida;

int main(void){
  Medida dist1;
  Medida dist2;
  Medida soma;

  printf("Primeira distância\n");

  // Entrada do campo metro para a variável estrutura dist1
  printf("Digite a quantidade de metros: ");
  scanf("%d", &dist1.metro);

  // Entrada do campo centimetro para a variável estrutura dist1
  printf("Digite a quantidade de centímetros: ");
  scanf("%d", &dist1.centimetro);
```

```
33   printf("Segunda distância\n");
34
35   // Entrada do campo metro para a variável estrutura dist2
36   printf("Digite a quantidade de metros: ");
37   scanf("%d", &dist2.metro);
38
39   // Entrada do campo centimetro para a variável estrutura dist2
40   printf("Digite a quantidade de centímetros: ");
41   scanf("%d", &dist2.centimetro);
42
43   soma.metro = dist1.metro + dist2.metro;
44   soma.centimetro = dist1.centimetro + dist2.centimetro;
45
46   if (soma.centimetro > 99){
47     ++soma.metro;
48     soma.centimetro -= 100;
49   }
50
51   printf("Soma das distâncias: %d m %d cm\n", soma.metro, soma.centimetro);
52
53   return 0;
54 } // main
```

Uso

```
> ./exemplo2.exe
Primeira distância
Digite a quantidade de metros: 2
Digite a quantidade de centímetros: 30

Segunda distância
Digite a quantidade de metros: 3
Digite a quantidade de centímetros: 60
Soma das distâncias: 5 m 90 cm

> ./exemplo2.exe
Primeira distância
Digite a quantidade de metros: 2
Digite a quantidade de centímetros: 30

Segunda distância
Digite a quantidade de metros: 3
Digite a quantidade de centímetros: 95
Soma das distâncias: 6 m 25 cm
```

Operações sobre estruturas

- Uma estrutura é uma entidade que pode ser tratada como um todo.
- Entretanto, somente podemos atribuir toda uma estrutura a outra.
  - Ou seja, uma estrutura pode somente ser copiada para uma outra estrutura do mesmo tipo usando o operador de atribuição.
- Se quisermos copiar todo o conteúdo da variável estrutura `ex1` para a variável `ex2`, simplesmente faremos

```
typedef struct {
  int x;
  int y;
  float t;
  char u;
} Exemplo;

Exemplo ex1 = {2, 5, 3.2, 'A'};
Exemplo ex2 = { 7, 3 };
ex2 = ex1;
```

- Após a execução da última linha acima, todos os respectivos campos de `ex1` e `ex2` possuirão os mesmos valores.

Apontadores para estruturas

- Estruturas, como outros tipos, podem também ser acessadas por meio de apontadores.
- Veja abaixo como usamos a estrutura Exemplo com apontadores:

```
typedef struct {
  int x;
  int y;
  float t;
  char u;
} Exemplo;

Exemplo ex1;
Exemplo *pExemplo;
...
pExemplo = &ex1;
```

- Podemos acessar a estrutura e todos seus campos usando o apontador `pExemplo`.
- A estrutura como um todo pode ser acessada usando indireção (*):

```
*pExemplo
```

Apontadores para estruturas

- Como o apontador contém o endereço do início da estrutura, podemos usá-lo para acessar cada campo da estrutura:

```
(*pExemplo).x  (*pExemplo).y  (*pExemplo).t  (*pExemplo).u
```

- Note que os parênteses são *absolutamente necessários* e omiti-los é um erro muito comum.
  - Se esquecermos os parentênses `*pExemplo.x` será interpretado como `*(pExemplo.x)` que é errado.
  - A expressão `*(pExemplo.x)` significa que temos uma variável estrutura chamada `pExemplo` que possui um campo `x` e esse deve ser um apontador.

Operador de seleção indireta

- Um outro operador, o de seleção indireta, elimina o problema com apontadores para estruturas.
- Ao invés de escrevermos

  ```
  (*nomeDoApontador).nomeDoCampo
  ```

  podemos usar

  ```
  nomeDoApontador->nomeDoCampo
  ```

- O operador de seleção indireta `->` poderia fazer referência a qualquer campo de uma estrutura. Abaixo vemos três formas de acessar o mesmo campo da estrutura `Exemplo`

```
Exemplo ex1;
Exemplo *pExemplo;
pExemplo = &ex1;

ex1.x = 1;        // o campo x de ex1 contém 1
(*pExemplo).x = 2; // Agora o campo x de ex1 contém 2
pExemplo->x = 3; // Por fim, o campo x de ex1 contém 3
```

Estruturas complexas

- As estruturas foram projetadas para lidar com problemas complexos.
- Podemos combiná-las com vários outros tipos de dados para modelar o problema.
- Por exemplo, podemos ter estruturas dentro de estruturas (estruturas aninhadas), arranjos dentro de estruturas, arranjos de estruturas, etc.

Estruturas aninhadas

- Podemos ter estruturas como membros de outra estrutura.
- Quando uma estrutura inclui uma outra estrutura, essa é chamada de estrutura aninhada.
- Não há limite no número de estruturas que podem ser aninhadas, mas é raro vê-las com mais de três níveis de aninhamento.
- Como exemplo podemos ter uma estrutura chamada `RegistroTemporal` que armazene a data e a horário do dia.
- A data deve guardar o dia, o mês e o ano.
- O horário do dia é uma estrutura que guarda a hora, os minutos e os segundos.
- A definição da estrutura aninhada `RegistroTemporal` é mostrada no próximo slide.

Estruturas aninhadas

```
typedef struct
{
   int dia;
   int mes;
   int ano;
} Data;

typedef struct
{
   int hora;
   int min;
   int seg;
} Horario;

typedef struct
{
   Data data;
   Horario horario;
} RegistroTemporal;

RegistroTemporal registro;
```

Estruturas aninhadas

- É possível aninhar estruturas do mesmo tipo mais que uma vez na declaração de uma estrutura.
- Como exemplo, considere uma estrutura que contenha o dia e o horário de início e de fim de uma tarefa.
- Podemos usar a estrutura `RegistroTemporal` duas vezes para criar uma nova declaração:

```
typedef struct
{
   RegistroTemporal inicio;
   RegistroTemporal fim;
} Tarefa;

Tarefa tarefa;
```

- Para acessar uma estrutura aninhada, incluímos cada nível desde o mais alto até o componente sendo referenciado.

Estruturas aninhadas

- Abaixo estão as formas de acessar as estruturas `RegistroTemporal` e `Tarefa`:

```
registro
registro.data
registro.data.dia
registro.data.mes
registro.data.ano
registro.horario
registro.horario.hora
registro.horario.min
registro.horario.seg

tarefa.inicio.horario.hora
tarefa.fim.horario.hora
```

Inicialização de estruturas aninhadas

- A inicialização de estruturas aninhadas segue as mesmas regras de uma estrutura simples.
  - Cada estrutrura deve ser completamente inicializada antes de seguir para o próximo campo.
  - Cada estrutura é envolta por chaves.
- Como exemplo, para inicializar `registro`, primeiro inicializamos `data` e depois `horario` separados por vírgula.
- Para inicializar `data` colocamos valores para `dia`, `mes` e `ano`, separados por vírgula.
- Podemos então inicializar os campos de `horario`.
- Uma definição e inicialização para `RegistroTemporal` é mostrada a seguir:

```
RegistroTemporal registro = {{7, 7, 2018}, {11, 48, 13}};
```

Estruturas contendo arranjos

- Estruturas podem ter um ou mais arranjos como membros.
- Os arranjos são acessados via indexação ou via apontadores, desde que qualificados corretamente com o operador de seleção direta.
- Abaixo está a definição de estrutura **`Estudante`** que contém o nome, as notas no semestre (3 no total) e a nota na prova final.

```
typedef struct
{
   char nome[30];
   int  nota[3];
   int  final;
} Estudante;

Estudante estudante;
```

- Os campos da estrutura acima podem ser referenciados por

```
estudante.nome
estudante.nome[i]
estudante.nota
estudante.nota[j]
estudante.final
```

Estruturas contendo arranjos

- Quando uma estrutura contém um arranjo podemos usar um apontador para se referir diretamente a um dos elementos do arranjo.
- Como exemplo podemos somar as notas do estudante usando

```
int *pNotas;
...
pNotas = estudante.nota;
notaTotal = *pNotas + *(pNotas + 1) + *(pNotas + 2);
```

- Aqui, `pNotas` aponta para a primeira nota, `pNotas + 1` para segunda, `pNotas + 2` para a terceira e assim, sucessivamente, se houvesse mais.
- A estrutura `Estudante` pode ser inicializada assim

```
Estudante estudante = {"Odair José", {92, 80, 70}, 87};
```

- Observe que o nome é inicializado como uma string e as notas são envoltas em chaves.

Arranjo de estruturas

- Existem situações onde precisamos criar um arranjo de estruturas.
- Por exemplo, poderíamos criar um arranjo de `estudantes` para trabalhar com um grupo de estudantes, cada um armazenado em uma estrutura.
- Ao colocar os dados no arranjo podemos, por exemplo, calcular rapidamente a média da turma.
- Como uma estrutura é um tipo, podemos criar um arranjo delas da mesma forma que criamos um arranjo de inteiros:

```
Estudante estudantes[67];
```

Arranjo de estruturas

- Vamos escrever um trecho de código que calcula a média da turma na prova final:

```
int notaTotal = 0;
float media;
Estudante *pEstud;
Estudante *pUltimoEstud;

pUltimoEstud = estudantes + 66; // endereço do último elemento
    do arranjo
for (pEstud = estudantes; pEstud <= pUltimoEstud; pEstud++)
   notaTotal += pEstud->final
media = notaTotal / 67.0;
```

- Se desejarmos acessar a nota do quinto estudante na segunda prova, usamos

```
estudantes[4].nota[1]
```

Estruturas e funções

- Em C, uma estrutura pode ser passada para uma função por dois métodos:
  1. Passagem por valor.
  2. Passagem por endereço.
- Nos próximos slides veremos exemplos destas duas formas de passagem.

Passando uma estrutura por valor

- Uma variável estrutura pode ser passada para uma função da mesma forma que as outras variáveis.
- Se a estrutura é passada por valor, alterações feitas na variável estrutura dentro da definição da função não seram refletidas na variável estrutura originalmente passada.
  - Ou seja, o que é passado é apenas uma cópia da estrutura original.

Exemplo 3 (Passagem de estrutura por valor para função)

O programa a seguir cria uma estrutura estudante, lê as informações do teclado e depois a passa para uma função que mostrará as informações lidas.

```
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13
14 typedef struct
15 {
16   char nome[30];
17   int  nota[3];
18   int  final;
19 } Estudante;
20
21
22 void imprimeEstudante(Estudante estudante){
23   printf("\nNome: %s\n", estudante.nome);
24
25   for(int i = 0; i < 3; i++)
26     printf("Nota %d: %d\n", i+1, estudante.nota[i]);
27   printf("Nota na prova final: %d\n", estudante.final);
28
29   return;
30 } // imprimeEstudante
```

```
33 int main(void){
34   Estudante estudante;
35
36   printf("Digite o nome do estudante: ");
37   if (fgets(estudante.nome, sizeof(estudante.nome), stdin) == NULL){
38       fprintf(stderr, "Erro ao ler o nome\n");
39       exit(1);
40     } else {
41       // retira o \n do final da string lida
42       estudante.nome[strcspn(estudante.nome, "\n")] = '\0';
43     }
44
45   printf("Digite as 3 notas: ");
46   scanf("%d %d %d", &estudante.nota[0], &estudante.nota[1],
47         &estudante.nota[2]);
48
49   printf("Digite a nota da prova final: ");
50   scanf("%d", &estudante.final);
51
52   imprimeEstudante(estudante);
53
54   return 0;
55 } // main
```

Uso

```
> ./exemplo3.exe
Digite o nome do estudante: Pedro Luiz
Digite as 3 notas: 14 27 9
Digite a nota da prova final: 20

Nome: Pedro Luiz
Nota 1: 14
Nota 2: 27
Nota 3: 9
Nota na prova final: 20
```

Passagem por referência

- O endereço de memória de uma estrutura é passado para uma função quando a passagem é por referência.
- Se uma estrutura é passada por referência, alterações feitas na variável estrutura dentro da função serão refletidas na variável estrutural original passada.

Exemplo 4 (Passagem de estrutura por referência para função)

O programa a seguir soma duas distâncias dadas em metros e centímetros e exibe o resultado.

```
10 #include <stdio.h>
11
12 typedef struct
13 {
14   int metro;
15   int centimetro;
16 } Medida;
17
18
19 void somaDistancias(Medida d1, Medida d2, Medida *pSoma){
20   pSoma->metro = d1.metro + d2.metro;
21   pSoma->centimetro = d1.centimetro + d2.centimetro;
22
23   if (pSoma->centimetro > 99){
24     ++pSoma->metro;
25     pSoma->centimetro -= 100;
26   }
27
28   return;
29 } // somaDistancias
```

```c
int main(void){
  Medida dist1;
  Medida dist2;
  Medida soma;

  printf("Primeira distância\n");
  // Entrada do campo metro para a variável estrutura dist1
  printf("Digite a quantidade de metros: ");
  scanf("%d", &dist1.metro);

  // Entrada do campo centimetro para a variável estrutura dist1
  printf("Digite a quantidade de centímetros: ");
  scanf("%d", &dist1.centimetro);

  printf("Segunda distância\n");
  // Entrada do campo metro para a variável estrutura dist2
  printf("Digite a quantidade de metros: ");
  scanf("%d", &dist2.metro);

  // Entrada do campo centimetro para a variável estrutura dist2
  printf("Digite a quantidade de centímetros: ");
  scanf("%d", &dist2.centimetro);

  somaDistancias(dist1, dist2, &soma);

  printf("Soma das distâncias: %d m %d cm\n", soma.metro, soma.centimetro);
  return 0;
} // main
```

Uso

```
> ./exemplo4.exe
Primeira distância
Digite a quantidade de metros: 2
Digite a quantidade de centímetros: 30

Segunda distância
Digite a quantidade de metros: 3
Digite a quantidade de centímetros: 90

Soma das distâncias: 6 m 20 cm
```

Note no programa anterior que tanto `dist1` quanto `dist2` são passadas por valor enquanto `soma` é passada por endereço para a função `somaDistancias`.

Uniões

- Uma união é uma construção que permite que a memória seja compartilhada por tipos diferentes de dados.
- Por exemplo, sabemos que um inteiro curto usa 2 bytes e cada caractere usa 1 byte. Portanto, poderíamos processar um inteiro curto como um número ou como dois caracteres.
- A união segue o mesmo formato que uma estrutura, exceto pelo presença da palavra `union` ao invés de `struct`.
- No exemplo a seguir é mostrado como declarar uma união que pode ser usada como um inteiro curto ou como dois caracteres.

```
union CompartilhaDados
{
   char vetC[2];
   short num;
};

union CompartilhaDados compartilhaDados;
```

Acessando membros de uma união

- As regras para acessar um membro de união são as mesmas de uma estrutura.
- Como exemplo, podemos acessar os dois campos da união `CompartilhaDados` assim

```
compartilhaDados.num
compartilhaDados.vetC[0]
```

Exemplo 5 (União entre short int e dois char)

O programa a seguir usa uma união para imprimir uma variável, primeiro como um número e depois como dois caracteres.

```c
#include <stdio.h>

typedef union
{
  short num;
  char vetC[2];
} CompartilhaDados;

int main(void){
  CompartilhaDados dados;

  dados.num = 16706;

  printf("Short: %hd\n", dados.num);
  printf("vetC[0]: %c\n", dados.vetC[0]);
  printf("vetC[1]: %c\n", dados.vetC[1]);

  return 0;
} // main
```

Uso

```
> ./exemplo5.exe

Short: 16706
vetC[0]: B
vetC[1]: A
```

Para se convencer que 16706 cria os caracteres A e B, você precisará analisar o padrão de bits desse número. No próximo slide mostrarei um programa para fazer isso.

Exemplo 6 (Examinando o padrão de bits de um número short int)

O programa desse exemplo imprime a representação binária de um dado número short int e dos caracteres A e B.

```c
#include <stdio.h>

void bin(unsigned short n, unsigned short numbits)
{
  unsigned short i;
  for (i = 1 << (numbits - 1); i > 0; i >>= 1)
    (n & i)? printf("1") : printf("0");

  return;
} // bin

int main(void){

  printf("Representação binária de %d\n", 16706);
  bin(16706, 16);
  printf("\n");

  printf("Representação binária de %c\n", 'B');
  bin('B', 8);
  printf("\n");

  printf("Representação binária de %c\n", 'A');
  bin('A', 8);
  printf("\n");

  return 0;
} // main
```

Uso

```
> ./exemplo6.exe
Representação binária de 16706
0100000101000010
Representação binária de B
01000010
Representação binária de A
01000001
```

Os processadores da Intel usam uma ordenação de bytes chamada de Little Endian. Isso significa que o byte de menor ordem do número é colocado em um endereço de memória menor que o byte de maior ordem. Assim, se o short int com 2 bytes deveria ser lido na ordem

```
Byte1 Byte0
```

na memória ele ficará assim

```
endereço inicial:   Byte0
endereço inicial+1: Byte1
```

Com isso o primeiro elemento de `vetC` coincide com `Byte0` e o segundo com `Byte1`.

Uniões e estruturas

- Em uma *union* cada parte dos dados inicia no mesmo endereço de memória e ocupa pelo menos uma parte da mesma memória.
- Quando uma união é compartilhada por dois ou mais tipos de dados diferentes, somente uma parte dos dados pode estar na memória no mesmo instante.
- Considere, por exemplo, uma agenda que contenha tanto o nome de uma empresa quanto nomes de pessoas.
  - Quando guardamos o nome da empresa, o nome em si possui apenas um campo.
  - Já quando guardamos os nomes de pessoas, o nome possui pelo menos três partes: o nome, o nome do meio e o sobrenome.
- Se quisermos ter apenas somente o campo nome na agenda precisamos usar uma *union*.
- A definição dessa entrada para a agenda está no próximo slide.

Agenda

```
typedef struct
{
   char prenome[20];
   char inicial;
   char sobrenome[30];
} Pessoa;

typedef struct
{
  char tipo;
  union
  {
     char empresa[40];
     Pessoa pessoa;
  } un;
} Entrada;
```

Note que no código **un** é o nome do campo para acessar uma *union*

Uniões e estruturas

- Como dois tipos diferentes de dados podem ser armazenados na mesma *union*, precisamos saber qual tipo está armazenado neste instante.
  - Isso é feito com o uso do campo `tipo` que é inicializado quando o dado for armazenado.
  - Se o nome for de um empresa o campo é E, já se for de uma pessoa o campo é P.
- Quando uma `union` é definida, C reserva espaço suficiente para armazenar o maior objeto de dados presente na união.
- No exemplo do slide anterior
  - O nome da empresa ocupa 40 bytes e o de uma pessoa, 51.
  - O tamanho da estrutura `Pessoa` que contém a *union* é 52 bytes, 51 bytes da união mais 1 para o tipo.
  - Quando o nome da empresa é armazenado os últimos 11 bytes estão presentes, mas não serão usados.
- No próximo slide é ilustrado o uso de `union` em uma estrutura.

Exemplo 7 (União como membro de uma estrutura)

Este exemplo demostra o uso de uma união encaixada em uma estrutura.

```
 9 #include <stdio.h>
10 #include <string.h>
11
12 typedef struct
13 {
14    char prenome[20];
15    char inicial;
16    char sobrenome[30];
17 } Pessoa;
18
19 typedef struct
20 {
21   char tipo;
22   union
23   {
24      char empresa[40];
25      Pessoa pessoa;
26   } un;
27 } Entrada;
```

```
30 int main(void){
31   Entrada negocio = {'E', "Empresa ABC"};
32   Entrada amigo;
33   Entrada agenda[2];
34
35   amigo.tipo = 'P';
36   strcpy(amigo.un.pessoa.prenome, "Marta");
37   strcpy(amigo.un.pessoa.sobrenome, "Silva");
38   amigo.un.pessoa.inicial = 'A';
39
40   agenda[0] = negocio;
41   agenda[1] = amigo;
42
43   for(int i = 0; i < 2; i++)
44     switch (agenda[i].tipo){
45     case 'E': printf("Empresa: %s\n", agenda[i].un.empresa);
46               break;
47     case 'P': printf("Amigo: %s %c. %s\n",
48          agenda[i].un.pessoa.prenome,
49          agenda[i].un.pessoa.inicial,
50          agenda[i].un.pessoa.sobrenome);
51               break;
52     default: printf("Erro no tipo\n"); break;
53     } // switch
54
55   return 0;
56 } // main
```

Uso

```
> ./exemplo7.exe
Empresa: Empresa ABC
Amigo:  Marta A. Silva
```

Exemplo 8 (Diferença entre união e estrutura – memória)

Neste exemplo veremos que mesmo uma união sendo similar a uma estrutura de várias formas, é importante saber a diferença entre as duas. Vamos definir uma união e uma estrutura com os mesmos campos e depois vamos medir e imprimir a quantidade de memória ocupada pelas duas.

```c
#include <stdio.h>

typedef union
{
  char nome[32];
  float salario;
  int cpf;
} UniaoTrab;

typedef struct
{
  char nome[32];
  float salario;
  int cpf;
} StructTrab;


int main(void){

  printf("espaço ocupado pela união: %5zu bytes\n", sizeof(UniaoTrab));
  printf("espaço ocupado pela estrutura: %5zu bytes\n", sizeof(StructTrab));

  return 0;
} // main
```

Uso

A saída do programa anterior é:

```
> ./exemplo8.exe

espaço ocupado pela união:     32 bytes
espaço ocupado pela estrutura: 40 bytes
```

- Como visto, há uma diferença na alocação de memória entre uma união e uma estrutura.
- A quantidade de memória necessária para armazenar uma estrutura é a soma do tamanho ocupado por todos os seus membros.
  - `nome` usa 32 bytes, `salario` usa 4 e `cpf` usa 4 bytes. Logo, a soma é 40 bytes
- Em uma união a quantidade de memória necessária é a quantidade de memória necessária para guardar o seu maior membro.
  - `nome` é o campo da união que necessita de mais espaço, 32 bytes. Logo, essa é quantidade de memória necessária para a união.

Exemplo 9 (Na união somente um membro pode ser acessado por vez)

Em uma estrutura todos os seus membros podem ser acessados em qualquer tempo. Entretanto, em uma união, somente um de seus membros pode ser acessado por vez e todos os outros conterão valores indefinidos.

```c
#include <stdio.h>

typedef union
{
  char  nome[32];
  float salario;
  int   cpf;
} UniaoTrab;


int main(void){
  UniaoTrab trab;

  printf("Digite o nome:\n");
  scanf("%s", trab.nome);

  printf("Digite o salário:\n");
  scanf("%f", &trab.salario);

  printf("\nMostrando\n");
  printf("Nome: %s\n", trab.nome);
  printf("Salário: %.1f\n", trab.salario);

  return 0;
} // main
```

Uso

A saída do programa anterior é:

```
> ./exemplo9.exe

Digite o nome:
Macunaima
Digite o salário:
1234.23

Mostrando
Nome: \G%Dnaima
Salário: 1234.2
```

Inicialmente, `Macunaima` será armazenado em `trab.nome` e todos os outros campos de `trab`, ou seja, `salario` e `cpf`, conterão valores indefinidos.

Entretanto, quando o valor do salário for pedido, 1234.23 será armazenado em `trab.salario` e todos os outros campos, ou seja, `nome` e `cpf`, agora terão valores indefinidos.

Para saber mais

- Forouzan, B. A and Gilbert, R. F. *Computer Science: a structured programming approach using C*. 3rd edition. Cengage Learning, 2007.

Fontes

- Forouzan, B. A and Gilbert, R. F. *Computer Science: a structured programming approach using C*. 3rd edition. Cengage Learning, 2007.
- *C Programming Enumeration*. Disponível em `https://www.programiz.com/c-programming/c-enumeration`.
- *C Programming Structure*. Disponível em `https://www.programiz.com/c-programming/c-structures`
- *How to pass structure to a function in C programming?*. Disponível em `https://www.programiz.com/c-programming/c-structure-function`.
- *C Programming Unions*. Disponível em `https://www.programiz.com/c-programming/c-unions`.