Estruturas de Dados I

Introdução

Prof. Bruno Azevedo

Instituto Federal de São Paulo

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Catanduva

Estruturas

- **Estruturas** são agrupamentos de variáveis referenciadas sob um nome.
- Fornecem um modo conveniente de ter informações relacionadas agrupadas.
- As variáveis que compõem a estrutura são geralmente chamadas de **campos** ou **elementos**.
- Estruturas são definidas utilizando a palavra-chave **struct** da seguinte forma:

```
struct nome_estrutura {
    //campos da estrutura
};
```

Estruturas

- Geralmente, os campos da estrutura são logicamente relacionados.
- Por exemplo, vamos definir uma estrutura para representar informações de endereço de casas.

```
struct endereco {
    string rua;
    int numero;
    string cidade;
    string estado;
    string cep;
};
```

- Entendam que ao definirmos uma estrutura não estamos declarando nenhuma variável.
- Apenas definimos a forma dos dados. Para declararmos essa estrutura precisamos fazer:

```
struct endereco informacoesDeEndereco;
```

- Isso declara uma variável **do tipo struct endereco** chamada de **informacoesDeEndereco**.

Estruturas

- Podemos também declarar uma ou mais variáveis na definição da estrutura.

```
struct endereco {
    string rua;
    int numero;
    string cidade;
    string estado;
    string cep;
} informacoesDeEndereco, iEndereco;
```

- Aqui definimos uma estrutura do tipo **endereco** e declaramos duas variáveis **do tipo struct endereco** chamadas de **informacoesDeEndereco** e **iEndereco**.

Estruturas Anônimas

- Podemos também definir uma estrutura anônima.

```
struct {
    string rua;
    int numero;
    string cidade;
    string estado;
    string cep;
} informacoesDeEndereco, iEndereco;
```

- Neste caso não conseguiremos declarar variáveis deste tipo em outros lugares do código.

Manipulando Campos de Estruturas

- Usamos o operador ponto "." para acessarmos e manipularmos os campos de uma estrutura.
- Considerando a estrutura abaixo:

```
struct {
    string rua;
    int numero;
    string cidade;
    string estado;
    string cep;
} informacoesDeEndereco, iEndereco;
```

- Se quisermos modificar o elemento **rua** na variável informacoesDeEndereco faremos:

```
informacoesDeEndereco.rua = "Novo nome de rua";
```

- Ou seja, a sintaxe para referenciar campos de uma estrutura é **nomeDeVariável.campo**.

Atribuições de Estruturas

- A informação contida em uma estrutura pode ser atribuída a outra estrutura de mesmo tipo.
- Ou seja, não é necessário atribuir cada elemento separadamente.
- Por exemplo:

```
struct {
    int a, b;
} e1, e2;
int main() {
    e1.a = 64;
    e1.b = 256;
    e2 = e1;
    return 0;
}
```

- Neste exemplo, a variável **e2** terá seus campos a e b definidos como 64 e 256, respectivamente.

Vetores de Estruturas

- Estruturas são tipos complexos (não-primitivos) de variáveis, mas se comportam como tipos primitivos em diversos aspectos.
- Por exemplo, podemos declarar vetores de estruturas.

```
struct Exemplo {
    int a, b;
};
int main() {
    Exemplo vetorDeExemplos[100];
    return 0;
}
```

- Acima, declaramos um vetor do tipo **Exemplo** de 100 elementos.

Vetores de Estruturas

- Os campos de um elemento de um vetor podem ser acessados e manipulados através de seu índice e do operador ponto.
- Por exemplo, se quisermos modificar o campo **a** do sétimo elemento do **vetorDeExemplos** faremos:

```
struct Exemplo {
    int a, b;
};
int main() {
    Exemplo vetorDeExemplos[100];
    vetorDeExemplos[6].a = 100;
    return 0;
}
```

Estruturas e Funções

- Podemos passar campos de estruturas como argumentos para funções.

```
struct Exemplo {
    int a, b;
} e1, e2;
int funcaoExemplo(int par1, int par2) {
    return par1 + par2;
}
int main() {
    e1.a = 64;
    e2.b = 256;
    cout << funcaoExemplo(e1.a, e2.b); // 320
    return 0;
}
```

Estruturas e Funções

- Podemos passar estruturas inteiras como argumentos para funções.

```
struct Exemplo {
    int a, b;
} e1;
int funcaoExemplo(Exemplo par1) {
    return par1.a + par1.b;
}
int main() {
    e1.a = 64;
    e1.b = 256;
    cout << funcaoExemplo(e1); // 320
    return 0;
}
```

Ponteiros para Estruturas

- A seguinte sintaxe permite declaramos ponteiros para estruturas: struct nomeDaEstrutura *nomeDaVariavel.

```
struct Exemplo {
    int a, b;
} e1;
int main() {
    struct Exemplo *e2;
    e2 = &e1;
    return 0;
}
```

- Também podem ser declarados na definição da estrutura.

```
struct Exemplo {
    int a, b;
} e1, *e2;
int main() {
    e2 = &e1;
    return 0;
}
```

- Ou seja, em muitos aspectos, podemos lidar com estruturas de modo semelhante a tipos primitivos.

Ponteiros para Estruturas

- Para acessarmos campos de estruturas através de ponteiros utilizamos o operador '->'.

```
struct Exemplo {
    int a, b;
} e1, *e2;
int main() {
    e2 = &e1;
    e2->a = 128;
    e2->b = 512;
    return 0;
}
```

Uniões

- Uma **união** é uma posição de memória compartilhada por duas ou mais variáveis diferentes.
- Geralmente são variáveis de **tipos diferentes**.
- Sua forma geral é semelhante à definição de estrutura.

```
union nome_união {
    // campos da união
};
```

- Por exemplo:

```
union tipo_u {
    int i;
    char c;
};
```

Uniões

- As variáveis **i** e **c** compartilham a mesma posição na memória.
- Cada uma ocupará um tamanho diferente (por exemplo, 4 bytes e 1 byte, respectivamente), mas ocuparão o mesmo endereço na memória.

```
union tipo_u {
    int i;
    char c;
};
```

- Quando uma união é declarada, o compilador reserva espaço suficiente para conter o **maior tipo** entre as variáveis da união.

Uniões

- Declaração de uniões, acesso e manipulação de campos, são efetuados de modo idêntico às estruturas.

```
union tipo_u {
    int i;
    char c;
} u1;
int main() {
    union tipo_u u2;
    u1.i = 1;
    u1.c = 'c';
    return 0;
}
```

- Um uso típico é para manipulação de dados heterogêneos. Em sistemas de arquivos ou protocolos de comunicação de rede, unions podem ser úteis para interpretar diferentes tipos de dados armazenados em uma mesma área de memória.
- Outro uso comum é para representar tipos variantes. Por exemplo, você pode usar uma união para representar uma variável que pode ser de um tipo inteiro, real ou caractere, dependendo do contexto.

Uniões

- Suponha que estejamos implementando um gerenciador de tabela de símbolos de um compilador.
- Um gerenciador de tabela de símbolos efetua o armazenamento de todas as variáveis, constantes, funções e outros símbolos encontrados durante a análise do código fonte.
- Os símbolos encontrados podem ser de vários tipos, incluindo inteiros, números de ponto flutuante ou sequências de caracteres. Por exemplo, 42, 3.14 ou "olá".
- Com uma união podemos ter uma única variável (a união) contendo qualquer um entre diversos tipos

```
union rótulo {
    int vali;
    float valf;
    chat *valc;
} u;
```

- É conveniente utilizar uma única variável para armazenar o símbolo, mas isso é um problema para linguagens fortemente tipadas como o C e o C++; uniões resolvem esse problema de modo elegante.
- E elas **garantem** que apenas um deles será utilizado em um determinado momento.

Uniões

- O que será exibido neste código?

```
union tipo_u {
    int i;
    char c;
} u1;
int main() {
    u1.i = 1;
    u1.c = 'c';
    cout << u1.i << " " << u1.c;
    return 0;
}
```

Uniões

- O que será exibido neste código?

```
union tipo_u {
    int i;
    char c;
} u1;
int main() {
    u1.i = 1;
    u1.c = 'c';
    cout << u1.i << " " << u1.c;
    return 0;
}
```

- Será exibido o valor 99 e o caractere 'c'. Será exibido o valor 99 e o caractere 'c'.
- O valor de u1.c é o valor esperado, que foi atribuído por último.

Uniões

- Não é complicado compreender porque o programa anterior exibe 99 e 'c'.
- Vamos supor que o espaço de memória alocado seja de 4 bytes, devido ao tamanho de i, e o espaço utilizado por c será de 1 byte.
- Inicialmente atribuímos 1 para i, portanto, temos o valor zero em todos os bits dos bytes, excetuando no bit menos significativo, que terá 1.
- Em seguida atribuímos 99 (110001) para o byte menos significativo. Portanto, o valor de i está com 99. Lembrem que o espaço de memória é **compartilhado** entre as variáveis.

Enumerações

- As enumerações são tipos de dados que permitem que você defina um conjunto de constantes com nomes simbólicos.
- São úteis quando desejamos representar um conjunto de valores discretos de forma legível e organizada.
- Uma enumeração é definida usando a palavra-chave ‘enum‘, seguida pelos nomes das constantes que você deseja definir.
- Cada constante é separada por vírgula, e o bloco de enumeração é finalizado por um ponto e vírgula.

```
enum DiasDaSemana {Domingo, Segunda, Terca, Quarta, Quinta, Sexta, Sabado};
```

- Cada constante acima está associada a um valor, iniciando com zero.
- Ou seja, “Domingo” é equivalente a 0 e “Sexta” é equivalente a 5.

Enumerações

- Por padrão, o primeiro valor em uma enumeração é 0 e cada valor subsequente é incrementado em 1. No entanto, você pode atribuir valores específicos a cada constante se desejar.

```
enum Meses {Janeiro = 1, Fevereiro = 2, Marco = 3, Abril = 4, Maio = 5,
            Junho = 6, Julho = 7, Agosto = 8, Setembro = 9, Outubro = 10,
            Novembro = 11, Dezembro = 12
};
```

- Uma vez definida a enumeração, você pode usar seus valores em seu código como se fossem constantes.
- Isso torna o código mais legível e evita o uso de números "mágicos".

Enumerações

```
#include <iostream>
using namespace std;
enum DiasDaSemana {
    Domingo,   // 0
    Segunda,   // 1
    Terca,     // 2
    Quarta,    // 3
    Quinta,    // 4
    Sexta,     // 5
    Sabado     // 6
};
int main() {
    DiasDaSemana dia = Sexta;
    if (dia == Segunda || dia == Terca || dia == Quarta || dia == Quinta || dia ==
       Sexta)
        cout << "Hoje é um dia de trabalho." << endl;
    else
        cout << "Hoje é um dia de descanso." << endl;
    return 0;
}
```

Exercícios de Estruturas

- **Cadastro de Alunos:** Crie uma estrutura `aluno` que contenha informações como nome, idade, número de matrícula e notas. Escreva um programa que permita cadastrar vários alunos e exiba suas informações.
- **Calculadora de Retângulos:** Defina uma estrutura `retângulo` que armazene as coordenadas do canto superior esquerdo e do canto inferior direito de um retângulo em um plano cartesiano. Escreva funções para calcular a área, o perímetro e o centro de massa de um retângulo.
- **Agenda de Contatos:** Crie uma estrutura `contato` que armazene informações como nome, telefone e email. Em seguida, implemente um programa que permita adicionar, remover, buscar e listar contatos.
- **Gerenciamento de Funcionários:** Defina uma estrutura `funcionario` que contenha informações como nome, cargo, salário e data de contratação. Escreva um programa que permita cadastrar funcionários e calcular o salário médio da equipe.
- **Biblioteca de Livros:** Crie uma estrutura `livro` que armazene informações como título, autor, ano de publicação e gênero. Desenvolva um programa que permita cadastrar livros, buscar por título ou autor e listar todos os livros disponíveis.

Exercícios de Uniões e de Enumerações

- **Cores:** Crie uma união que represente cores em diferentes formatos: valor RGB, valor RGBA, valor hexadecimal, string representando o nome da cor. Os dois primeiros devem ser estruturas, o terceiro deve ser um inteiro (o valor hexadecimal terá que ser convertido), e o quarto será uma string ou vetor de caracteres. Crie todos esses elementos e armazene uma cor em sua união.
- **Menu:** Crie um programa que simule um menu com opções numeradas. Utilize uma enumeração para representar as opções disponíveis no menu. O programa deve exibir o menu para o usuário e solicitar que ele selecione uma opção digitando o número correspondente. O menu deve conter as seguintes opções:
  1. Visualizar perfil
  2. Editar configurações
  3. Sair

  Depois que o usuário selecionar uma opção, o programa deve imprimir a opção selecionada. Se o usuário digitar um número que não corresponda a uma opção válida, o programa deve exibir uma mensagem de erro.

  Deve obrigatoriamente utilizar os elementos da enumeração para selecionar opções em seu código.