Tipos Compostos de Dados

# UNIÕES E ENUMERAÇÕES

Judson Santiago

# Introdução

- As variáveis e constantes armazenam informações
  - Elas ocupam espaço na memória
  - Possuem um tipo
- Os tipos básicos armazenam valores:

Inteiros

```
char   ch   = 'W';
short  sol  = 25;
int    num  = 45820;
```

Ponto-flutuantes

```
float  taxa = 0.25f;
double peso = 1.729156E5;
```

# Introdução

- Porém, com os tipos básicos não é possível armazenar um conjunto de informações
  - Como armazenar o peso de 22 jogadores?

```
float p1  = 80.2;
float p2  = 70.6;
float p3  = 65.5;
...
float p21 = 85.8;
float p22 = 91.0;
```

Criar 22 variáveis diferentes não é a melhor solução.

  - A solução é usar vetores:

```
float peso[22];
```

# Introdução

- Com vetores não é possível armazenar um conjunto de informações de tipos diferentes
  - Como armazenar um cadastro completo de 22 jogadores? (nome, idade, altura, peso, gols, etc.)

```
char nome[22][80];
unsigned idade[22];
unsigned altura[22];
float peso[22];
unsigned gols[22];
```

Criar vários vetores não é a melhor solução.

  - A solução é usar registros

# Introdução

- O registro agrupa informações, de tipos possivelmente diferentes, sob um único identificador

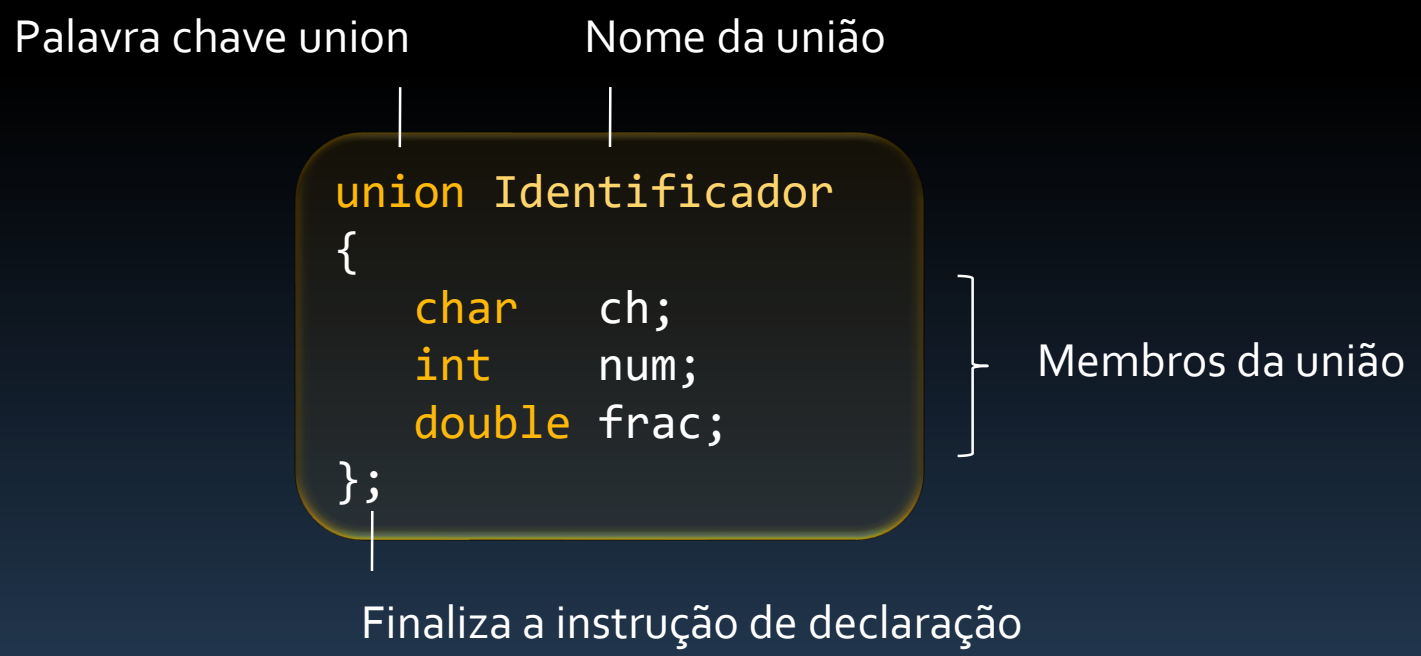
```
struct Jogador
{
    char nome[20];
    unsigned idade;
    unsigned altura;
    float peso;
    unsigned gols;
};
```

Com registros podemos criar um vetor do tipo jogador

```
Jogador equipe[22];
```

# Uniões

- Assim como um registro, uma união pode armazenar diferentes tipos de dados



# Uniões

- A diferença entre um registro e uma união é que a união só armazena um de seus membros por vez
  - O registro armazena um char e um int **e** um double
  - A união armazena um char **ou** um int **ou** um double

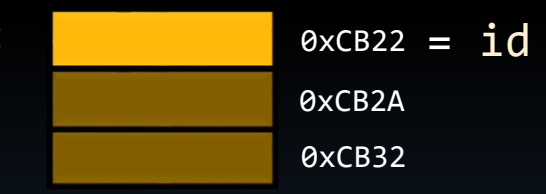
```
struct Identificador
{
    char   ch;
    int    num;
    double frac;
};
```

```
union Identificador
{
    char   ch;
    int    num;
    double frac;
};
```

# Uniões

- Os membros compartilham a mesma posição de memória
  - O tamanho do bloco é igual ao do maior membro

```
union Identificador
{
    char   ch;
    int    num;
    double frac;
};

Identificador id;
```



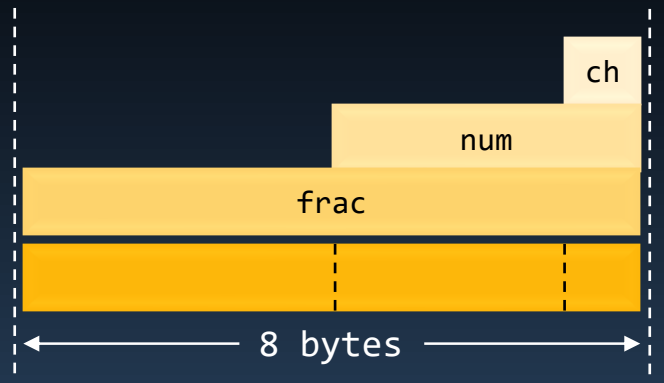

`id` ocupa 8 bytes, o mesmo tamanho de um double

# Uniões

- A memória é compartilhada entre os membros

```
Identificador id;

id.ch = 'a';       // char
cout << id.ch;     // a

id.num = 972;      // int
cout << id.num;    // 972

id.frac = 3.8;     // double
cout << id.frac;   // 3.8

cout << id.num;    // lixo
cout << id.ch;     // lixo
```

```
union Identificador
{
    char   ch;
    int    num;
    double frac;
};
```



# Uniões

```cpp
#include <iostream>
using namespace std;

union CharInt
{
    int num;
    char ch;
};

int main()
{
    CharInt val = {0};

    cout << "Digite um caractere: ";
    cin >> val.ch;
    cout << "Código ASCII: ";
    cout << val.num << endl;
}
```

CharInt.cpp

# Uniões

- Saída do Programa:

```
Digite um caractere: T
Código ASCII: 84
```

- A inicialização deve fornecer apenas um valor
  - O valor inicializa sempre o primeiro elemento

```
union CharInt
{
    int num;
    char ch;
};
```

```
CharInt val = {0,0};
```

```
CharInt val = { 0 };
```

# Uniões

- A união é usada para economizar memória
  - Quando um item pode usar dois ou mais formatos
  - Mas nunca ao mesmo tempo

Ex.: o número serial de um software pode ser uma chave inteira ou um código de caracteres

```
union Chave
{
   int  numero;
   char codigo[4];
};
```

```
struct Software
{
    char   nome[40];
    float  preco;
    Chave  serial;
    bool   tipo;
};
```

# Uniões

```cpp
#include <iostream>
using namespace std;

union Chave
{
    int  numero;
    char codigo[4];
};

int main()
{
    cout << "Qual seu tipo de chave?\n[1] número\n[2] código\nOpção: ";
    int tipo;
    cin >> tipo;

    Chave serial;
    if (tipo == 1) { cout << "Digite seu número: ";  cin >> serial.numero; }
    else { cout << "Digite seu código: "; cin >> serial.codigo; }
}
```

RegKey.cpp

# Uniões

- Saída do Programa:

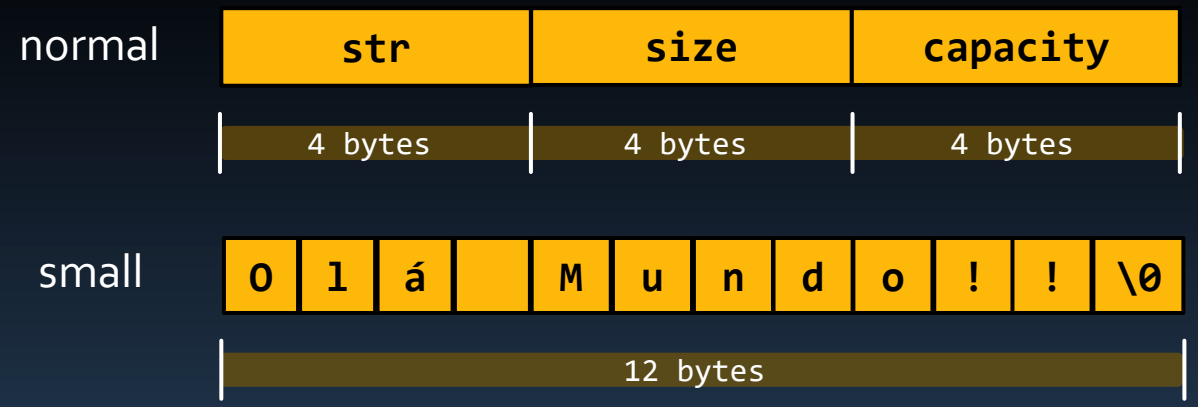
```
Qual seu tipo de senha?
[1] número
[2] código
Opção: 1
Digite seu número: 12508
```

- O programador só pode armazenar valores em um dos membros da união, portanto ele deve saber que informação foi digitada

# Uniões

- O tipo string usa uniões para implementar uma otimização para strings pequenas

```
string nome;
```




```
struct string
{
    union
    {
        struct
        {
            char * str;
            int size;
            int capacity;
        }
        normal;

        char small[12];
    }
    data;
    bool type;
};
```

# Enumerações

- Uma enumeração consiste em um conjunto de constantes inteiras, em que cada uma é representada por um nome

```
enum Cores {Verde, Amarelo, Azul, Branco, Preto};
```

- A instrução acima faz duas coisas:
  - Define Cores como o nome de um novo tipo
  - Faz dos nomes Verde, Amarelo, Azul, Branco e Preto constantes para os valores 0, 1, 2, 3 e 4

# Enumerações

- Fornece uma forma rápida de criar várias constantes

```
enum Cores {Verde, Amarelo, Azul, Branco, Preto};
```

  - A enumeração acima equivale as seguintes declarações:

```
const int Verde = 0;
const int Amarelo = 1;
const int Azul = 2;
const int Branco = 3;
const int Preto = 4;
```

# Enumerações

- Ela é usada quando conhecemos o conjunto de valores que uma variável pode assumir e desejamos usar nomes para esses valores dentro do programa

```
// P = 0, M = 1, G = 2
enum Tamanhos {P, M, G};

// Norte = 0, Sul = 1, Leste = 2, Oeste = 3
enum Direcao {Norte, Sul, Leste, Oeste};

// Vermelho = 0, Amarelo = 1, Verde = 2, Azul = 3, Preto = 4
enum Cores {Vermelho, Amarelo, Verde, Azul, Preto};
```

# Enumerações

```cpp
#include <iostream>
#include <random>
using namespace std;

enum Moeda { Cara, Coroa };

int main()
{
    cout << "Jogando a moeda...\n";

    random_device rand;
    int sorteio = rand() % 2;

    if (sorteio == Cara)
        cout << "Fica com a bola!\n";
    if (sorteio == Coroa)
        cout << "Escolhe o lado!\n";
}
```

Moeda.cpp

# Enumerações

- Saída do Programa:

```
Jogando a moeda...
Fica com a bola!
```

- O uso das constantes deixa o código mais claro que:

```
if (sorteio == 0)
    cout << "Fica com a bola!\n";
if (sorteio == 1)
    cout << "Escolhe o lado!\n";
```

# Enumerações

- Se a intenção é criar apenas constantes sem ter um tipo:

```
enum {Vermelho, Amarelo, Verde, Azul, Preto};
```

- Valores podem ser explicitamente definidos:

```
enum Bits {Um=1, Dois=2, Quatro=4, Oito=8};
```

- Alguns valores podem ser omitidos:

```
enum Bigstep {Primeiro, Segundo=100, Terceiro};
```

- Valores podem ser repetidos:

```
enum {Zero, Nulo=0, One, Um=1};
```

# Enumerações

- Após a definição da enumeração, é possível criar variáveis:

```
enum Cores {Vermelho, Amarelo, Verde, Azul, Preto};
Cores tinta;
```

  - As únicas atribuições válidas são as de um dos valores definidos na enumeração:

```
   tinta = Azul;        // válido
x  tinta = 2000;        // inválido
x  tinta = 3;           // inválido

   tinta = Cores (3);   // válido, type cast estilo C++
   tinta = (Cores) 3;   // válido, type cast estilo C
   int a = Azul;        // válido, Azul é uma constante inteira
```

# Enumerações

```cpp
#include <iostream>
using namespace std;

enum Mes {Jan=1, Fev, Mar, Abr, Mai, Jun, Jul, Ago, Set, Out, Nov, Dez};

int main()
{
     Mes inicio, fim; // cria variáveis do tipo mês

     inicio = Fev;    // inicio do ano letivo
     fim    = Nov;    // fim do ano letivo

     cout << "Digite o número do mês atual: ";
     int atual;
     cin >> atual;    // lê o mês atual para uma variável inteira

     if (atual >= inicio && atual <= fim)
          cout << "Você está em período de aulas.\n";
     else
          cout << "Férias!\n";
}
```

Meses.cpp

# Enumerações

- Saída do Programa:

```
Digite o mês atual: 3
Você está em período de aulas.
```

- As funções de entrada e saída (cin e cout) não sabem como ler ou mostrar um tipo definido pelo programador:

```
    cout << "Digite o mês atual: ";
    Mes atual;
x   cin >> atual; // cin não conhece o tipo Mes
```

A não ser que sejam ensinadas a fazer isso

# Enumerações com Escopo

- As enumerações tradicionais tem alguns problemas:
  - Duas definições podem ter nomes conflitantes:

```
enum Pacote { Pequeno, Grande, Largo, Jumbo};
enum Camisa { Pequena, Media, Grande, Extragrande };
```

  - O tipo de um enumerador é dependente da implementação:
    - Eles podem ser constantes de qualquer tipo inteiro
    - Embora, a partir do C++11, seja permitido especificar o tipo

```
enum Direcao : short {Norte, Sul, Leste, Oeste};
```

# Enumerações com Escopo

- As enumerações tradicionais tem alguns problemas:

```
enum Cores {Vermelho, Amarelo, Verde, Azul, Preto};
```

  - Enumeradores são implicitamente convertidos para inteiros:
    - Em atribuições

```
// converte vermelho para 0
int num = Vermelho;
```

    - Em comparações

```
// converte preto para 4
if (num < Preto)
```

# Enumerações com Escopo

- C++11 resolveu estes problemas com uma nova forma de enumeração que fornece escopo aos enumeradores
  - Enumeradores são de tipo int (quando o tipo não é indicado)

```
enum class Pacote { Pequeno, Grande, Largo, Jumbo};
enum class Camisa { Pequena, Media, Grande, Extragrande };

Pacote leite = Pacote::Grande;
Camisa promo = Camisa::Grande;

x int tamanho = Camisa::Media;      // conversão implícita não permitida
  int carga = int(Pacote::Jumbo);   // ok, conversão explícita
```

# Resumo

- Uniões são semelhantes a registros, mas só armazenam um membro por vez
  - Elas são usadas para economizar memória
  - Especialmente útil em grandes quantidades (vetores)
- Enumerações são usadas para definir constantes inteiras
  - É mais fácil trabalhar com nomes do que com números
  - São usadas quando o número de valores que uma variável pode assumir é conhecido e pequeno