# Minicurso de C++

## Aula 02
## Funções, Ponteiros e Estruturas de Dados

Responsáveis:
Gabriel Daltro Duarte
Rafael Guerra de Pontes
Walisson da Silva
Weslley da Cunha Santos
Ianes Grécia


# Funções

# Funções

- No geral, programas mais complexos exigem algoritmos complexos, mas sempre é possível dividir um problema grande em problemas menores.

- Cada uma dessas partes divididas também pode ser novamente dividida em partes mais simples de serem entendidas. Essa estratégia é conhecida como Dividir para Conquistar.

# Funções

- Cada parte menor do algoritmo complexo pode ser vista, em programação, como uma sub-rotina (uma função);

- As funções permitem que o programador modularize um programa separando suas tarefas em unidades autocontidas.

# Funções - Vantagens

- Técnica **dividir para conquistar.**
- Facilitam o projeto, implementação, operação e a manutenção de grandes programas.
- Reusabilidade do código ⇒ mesmas funções podem ser usadas em outros programas.
- Também chamadas de métodos ou procedimentos.
- Podem conter parâmetros.
- Podem retornar algum valor ou só retornar o controle à função que a chamou.

# Chamada de Função

- As funções são invocadas através de uma chamada de função.

- Quando uma função completa sua tarefa, ele pode retornar um resultado, ou  um sinal de controle para o chamador ou simplesmente não retornar nada;

```cpp
// Exemplos de funções da biblioteca cmath
#include <iostream>
#include <cmath>
using std::cout;
using std::endl;
int main(){
    const double e = std::exp(1.0);
        cout << "ceil(2.5) = " << ceil(2.5) << endl;
        cout << "floor(2.5) = " << floor(2.5) << endl;
        cout << "cos(M_PI/2) = " << cos(M_PI/2) << endl;
        cout << "cos(0) = " << cos(0) << endl;
        cout << "sin(M_PI/2) = " << sin(M_PI/2) << endl;
        cout << "sin(0) = " << sin(0) << endl;
        cout << "fabs(-12.5) = " << fabs(-12.5) << endl;
        cout << "exp(1) = " << exp(1) << endl;
        cout << "log(e) = " << log(e) << endl;
        cout << "log(2) = " << log(2) << endl;
        cout << "log10(10) = " << log10(10) << endl;
        cout << "log10(1000) = " << log10(1000) << endl;
        cout << "pow(2,10) = " << pow(2,10) << endl;
        return 0;
}
```

## Saída

```
ceil(2.5) = 3
floor(2.5) = 2
cos(M_PI/2) = 6.12323e-017
cos(0) = 1
sin(M_PI/2) = 1
sin(0) = 0
fabs(-12.5) = 12.5
exp(1) = 2.71828
log(e) = 1
log(2) = 0.693147
log10(10) = 1
log10(1000) = 3
pow(2,10) = 1024
```

# Declarando funções através de Protótipos

**Uma declaração de uma função dá o nome da função, o tipo de valor devolvido pela função (se houver) e o número e os tipos dos argumentos que precisam ser fornecidos em uma chamada para uma função.**

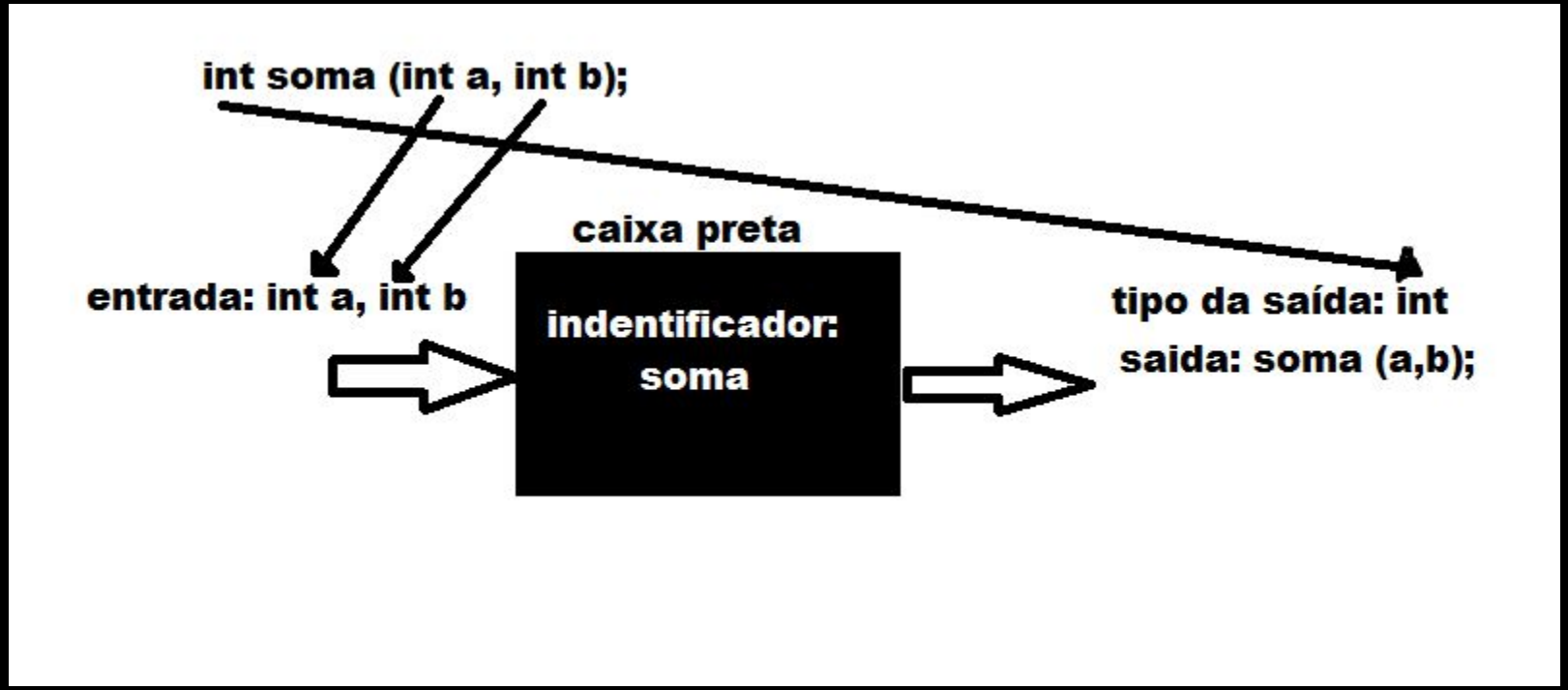
**Forma geral de um protótipo de função que recebe dois argumentos:**

```
tipo_de_retorno identificador_funcao (tipo_parametro0 parametro0, tipo_parametro1 parametro1);
```

Exemplos concretos:

```
int soma (int a, int b);
float quadrado (float n);
unsigned long long int fatorial (int n);
```

# Visão modular de uma função



# Definições de funções

Cada função que é chamada em um programa deve ser definida em algum lugar (uma vez);

A maneira que você especifica como uma operação deve ser feita é definindo uma função para fazê-lo;

```cpp
// Exemplo da definição da função que retorna a soma de dois inteiros

#include <iostream>

using namespace std;

int soma(int a, int b){
    return (a + b);
}

int main(){

    cout << "soma(2,3) = " << soma(2,3) << endl;

    return 0;
}
```

**Saída**

```
soma(2,3) = 5
```

# Exercício

Crie uma função que receba o valor do raio de uma esfera e, com isso, calcule e retorne o seu volume (Dado: $V_E = (4/3) * \pi * R^3$).

# Exercício para pensar

Como seria uma função que calcula o fatorial de um número?

```cpp
// Exemplo de função que calcula o fatorial de um número
#include <iostream>

using namespace std;

typedef unsigned long long int ulli;

ulli fatorial(int n){
    if(n == 0) return 1;
    ulli resultado = 1;
    for(int i = 1; i <= n; i++)
        resultado *= i;
    return resultado;
}

int main(){
    for(int i = 0; i <= 10; i++){
        cout << "fatorial(" << i << ") = ";
        cout << fatorial(i) << endl;
    }
     return 0;
}
```

**Saída**

```
fatorial(0) = 1
fatorial(1) = 1
fatorial(2) = 2
fatorial(3) = 6
fatorial(4) = 24
fatorial(5) = 120
fatorial(6) = 720
fatorial(7) = 5040
fatorial(8) = 40320
fatorial(9) = 362880
fatorial(10) = 3628800
```

# Classe de armazenamento

- Identificadores (variáveis e funções);

- A classe de armazenamento é o atributo de um identificador; ela ajuda a determinar a sua duração (tempo) de armazenamento, escopo e ligação.

- Especificadores de classes de armazenamento: **auto**, **extern** e **static**.

# Classe de armazenamento

- **auto**: tempo de armazenamento automático.
- Existem enquanto o bloco estiver ativo e são destruídas quando o bloco é deixado para trás (variáveis locais).
- Apenas variáveis podem ter tempo de armazenamento automático.
- As variáveis locais recebem essa classe de armazenamento por *default*.

# Classe de armazenamento

- **extern**: Usado para declarar variáveis ou funções que passarão a existir a partir do instante em que o programa for iniciado.
- As variáveis globais e nomes de funções são pertencentes à essa classe de armazenamento por *default*.
- Podem ser referenciados em qualquer função que venham após suas declarações no arquivo.

# Classe de armazenamento

- **static**: Variáveis desse classe são conhecidas apenas na função na qual foi definida, mas conservam o seu valor quando a função for encerrada.
- Todas as variáveis numéricas do tipo **static** são inicializadas com 0, se não forem inicializadas explicitamente.

# Classe de armazenamento

- Cuidado para não confundir classe de armazenamento com escopo!
- Por exemplo, variáveis **static** ainda tem escopo de bloco e não escopo de arquivo.

```cpp
#include <iostream>

int x; // variável global com escopo de arquivo e classe de armazenamento extern
void f_teste (void);

int main ()
{
    f_teste();
    f_teste();
    f_teste();
    x = 5;
    std::cout << std::endl << std::endl << "Valor final de X: " << x;
    return 0;
}

void f_teste (void)
{
    static int cont_static = 0;
    int cont_auto = 0;
    x = 2;
    cont_auto++;
    cont_static++; // cont existe durante toda a execução do programa, mas só é acessível dentro de f_teste();
    std::cout << "Valor de X: " << x << std::endl;
    std::cout << "Segundo cont_static: f_teste() foi chamada " << cont_static << " vezes" << std::endl;
    std::cout << "Segundo cont_auto: f_teste() foi chamada " << cont_auto << " vezes" << std::endl << std::endl;
}
```

# Sobrecarga de funções

- O C++ permite que várias funções do mesmo nome sejam definidas, contanto que essas funções tenham conjuntos diferentes de parâmetros no que diz respeito aos tipos de parâmetros, número de parâmetros ou a ordem dos tipos dos parâmetros.

- Essa capacidade é chamada sobrecarga de funções.

# Sobrecarga de funções

- Quando uma função sobrecarregada é chamada, o compilador C++ avalia o número, o tipo e a ordem dos parâmetros para que a função correta seja chamada.
- A sobrecarga de funções é comumente utilizada para criar várias funções do mesmo nome que realizam tarefas semelhantes, mas em tipos de dados diferentes.

```cpp
/* Exemplo de sobrecarga de funções */

#include <iostream>

int quadrado (int x)
{
    std::cout << "O quadrado do inteiro " << x << "eh" ;
    return x*x;
}


float quadrado (float x)
{
    std::cout << "O quadrado do float" << x << "eh ";
    return x*x;
}

int main ()
{
     std::cout << quadrado (7);
     std::cout << std::endl;
     std::cout << quadrado ((float) 7.5);
     return 0;
}
```

# Funções com argumentos padrão

- É comum um programador invocar uma função repetidamente e sempre passar os mesmos valores de argumentos para um parâmetro em particular;
- Nesses casos, o programador pode especificar um valor que será passado automaticamente a uma função, caso durante a chamada da função, o programa omita um argumento para parâmetro com argumento padrão;

# Funções com argumentos padrão

```
/* Exemplo de uma função que possui argumentos padrão */
#include <iostream>

int boxVolume (int comp = 0, int larg = 0, int altura = 0)
{
    return comp*larg*altura;
}

int main ()
{
    int vol1, vol2;
    vol1 = boxVolume ();
    vol2 = boxVolume (1,2,3);
    return 0;
}
```

# Ponteiros

# Ponteiros

- Variável cujo valor é um endereço de memória.
- Para que usar ponteiros?
  - Passagem de variáveis por referência;
  - Alocação dinâmica de memória;
  - Manipulação de arrays;
  - Usados para construir referências a estruturas de dados como listas encadeadas, árvores, grafos, etc.

# Ponteiros

- Variável cujo valor é um endereço de memória.
- Declaração:

```
int *ponteiroInteiro;
float *ponteiro_para_float;
```

- Operadores importantes:
  - operador de endereço (&)
  - operador de indireção ou desreferência (*)

# Exemplo

```cpp
#include <iostream>

using std::cout;
using std::endl;

int main(){
     int n; // n é um inteiro
     int *nPtr; // nPtr é um ponteiro para inteiro

     n = 123; // n recebe o valor 123
     nPtr = &n; // nPtr recebe o endereço de n

     cout << "O endereço de n é " << &n;
     cout << "\nO valor de nPtr é " << nPtr;
     cout << "\n\nO valor de n é " << n << endl;
     cout << "O valor de *nPtr é " << *nPtr;

     return 0;
}
```

# Ponteiros

- Caso não haja endereço imediato, atribuir NULL ao ponteiro por segurança.
- NULL é uma constante com valor 0.
- Geralmente, programas não podem acessar o endereço de memória 0, pois é reservado ao SO.
- Isso evita uso indevido da memória.

# Aritmética de Ponteiros

- Ponteiros dão suporte às seguintes operações aritméticas: ++, --, + e -.
- Dessa forma, pode-se navegar por posições adjacentes de memória, de acordo com o tipo de dado apontado.

# Ponteiros e Constantes

`const char *pParaConstante;` ⇒ **Aponta para um dado constante**

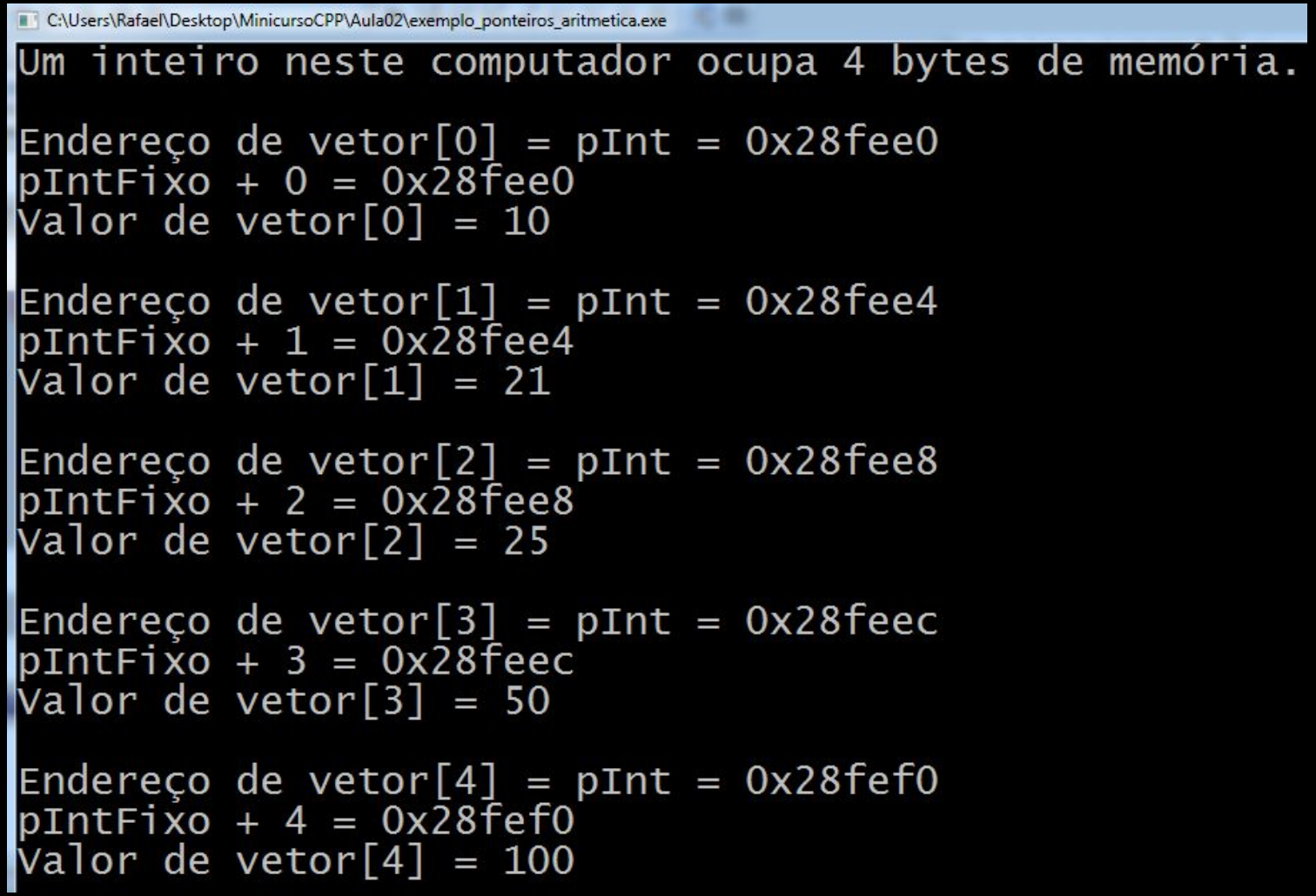
`char * const pConstante = &x;` ⇒ **Ponteiro não pode apontar para um endereço de memória diferente**

`const char * const pCparaC;` ⇒ **Ponteiro não pode apontar para endereço diferente nem pode modificar o conteúdo do endereço apontado.**

```cpp
// Exemplo de aritmética de ponteiros (++)
#include <cstdlib>
#include <iostream>
using namespace std;
int main(){
    system("chcp 1252");
    system("cls");
    int vetor[5] = {10,21,25,50,100};
    int *pInt;
    int * const pIntFixo = vetor;
    cout << "Um inteiro neste computador ocupa ";
    cout << sizeof(int) << " bytes de memória.\n\n";
    pInt = vetor; // pInt = &vetor[0];

    for(int i = 0; i < 5; i++){
        cout << "Endereço de vetor[" << i << "] = ";
        cout << "pInt = " << pInt << endl;
        cout << "pIntFixo + " << i << " = " << (pIntFixo+i) << endl;
        cout << "Valor de vetor[" << i << "] = ";
        cout << *pInt << "\n\n";
        pInt++; // Incrementa para a próxima posição
    }
    return 0;
}
```

Saída no próximo slide!


C:\Users\Rafael\Desktop\MinicursoCPP\Aula02\exemplo_ponteiros_aritmetica.exe
Um inteiro neste computador ocupa 4 bytes de memória.
Endereço de vetor[0] = pInt = 0x28fee0
pIntFixo + 0 = 0x28fee0
Valor de vetor[0] = 10
Endereço de vetor[1] = pInt = 0x28fee4
pIntFixo + 1 = 0x28fee4
Valor de vetor[1] = 21
Endereço de vetor[2] = pInt = 0x28fee8
pIntFixo + 2 = 0x28fee8
Valor de vetor[2] = 25
Endereço de vetor[3] = pInt = 0x28feec
pIntFixo + 3 = 0x28feec
Valor de vetor[3] = 50
Endereço de vetor[4] = pInt = 0x28fef0
pIntFixo + 4 = 0x28fef0
Valor de vetor[4] = 100

# Ponteiros vs. Arrays

- Um array é, na verdade, um ponteiro constante para um endereço de memória fixo que indica o início do array.
- Aritmética de ponteiros é válida para arrays.
- Não se pode modificar o endereço para o qual o array aponta.
  - Não é possível fazer `vetor++ ou vetor--`, por exemplo, caso `vetor` seja um array declarado.

# Exercício

Faça um programa que:

a) Declare uma variável x do tipo int e outra px do tipo ponteiro para int.

b) Atribua o valor 10 para x e o endereço de x para px.

c) Exiba o conteúdo e o endereço de x de forma direta (usando a própria variável).

d) Exiba o conteúdo e o endereço de x de forma indireta (usando o ponteiro).

# Tipos de Passagem de argumentos para funções

1 - Passagem por valor: Uma cópia do valor do argumento é feita e passada para a função chamada. As alterações na cópia não alteram o valor da variável original. (Exemplo: função soma mostrada anteriormente)

2 - Passagem por referência: O chamador fornece a função a capacidade de acessar os dados do chamador diretamente e modificá los se a função chamada escolher fazer isso.

# Passagem por referência com Ponteiros

- Para passar um ponteiro para uma função basta declará-lo como paramêtro do tipo ponteiro.
- Desse modo, o conteúdo da variável original poderá ser alterado diretamente pelo ponteiro.

# Exemplo

```
#include <iostream>

using namespace std;

void troca(int *valor);

int main()
{
      int numero = 10;
      troca(&numero);
      cout << numero << endl;

}

void troca(int *valor)
{
      *valor = 3;
}
```

# Exemplo

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

int soma(int *vetor,int tamanho);

int main()
{
      int numeros[5] = {10,20,30,40,50};
      int resultado = soma(numeros,5);
      cout << resultado << endl;

}
int soma(int *vetor,int tamanho)
{
      int i,soma=0;
      for(i=0;i<tamanho;i++)
      {
            soma+=vetor[i];
      }
      return soma;
}
```

# Exercício

Escreva uma função chamada troca, que receba o endereço de duas variáveis inteiras e troque o valor delas, um pelo outro. Escreva um programa para testar a função, exibindo as variáveis antes e depois da troca.

# Variáveis de Referência

- É uma variável com um outro nome para acessar uma variável já existente.
- Não existem referências nulas (NULL).
- Uma variável de referência SEMPRE deve ser atribuída a um endereço de memória legítimo.
- Deve ser inicializada na declaração.
- Não pode mudar sua referência ao longo do código.

```cpp
#include <iostream>
#include <cstdlib>

using namespace std;

int main (){
    // declaração simples de variáveis
    int    i;
    double d;
    system("chcp 1252");
    system("cls");
    // declaração de variágeis de referência
    int&    ir = i;
    double& rd = d;

    i = 5;
    cout << "Valor de i: " << i << endl;
    cout << "Valor da referência de i: " << ir  << endl;

    d = 11.7;
    cout << "Valor de d: " << d << endl;
    cout << "Valor da referência de d: " << rd  << endl;

    return 0;
}
```

## Saída

```
Valor de i: 5
Valor da referência de i: 5
Valor de d: 11.7
Valor da referência de d: 11.7
```

# Passagem por referência através de parâmetros de referência

- Deve seguir todas as regras de uma variável de referência.

```cpp
#include <iostream>

using namespace std;

void troca(int &valor);

int main()
{
	int numero = 10;
	troca(numero);
	cout << numero << endl;

}

void troca(int &valor)
{
	valor = 3;
}
```

# Estruturas de Dados

# Estruturas de Dados

- Arrays permitem armazenar conjuntos de dados do mesmo tipo.
- Estruturas permitem a definição de um tipo de dado abstrato definido pelo programador.

# Estruturas de Dados

- Definição:

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

# Estruturas de Dados

- Exemplo

```
struct Pessoa{
    int idade;
    string nome;
    float altura;
    double peso;
};
```

# Estruturas de Dados

- A “structure tag” é opcional, a não ser que seja necessária uma auto-referência.
- É possível declarar variáveis do tipo do **`struct`** definido imediatamente antes do `;` após sua declaração.