Curso Linguagem C++

Capítulo 7
Iniciando orientação a objetos

Passando estruturas como argumento

- Os problemas apontados acima só serão adequadamente resolvidos em **C++**. Mas há um outro problema que, mesmo em **C,** poderia ser resolvido de modo adequado: a maneira como a variável estruturada está sendo passada para as funções, pois essa é uma maneira ineficiente.

- Para entender porque, vamos analisar qualquer uma das funções que estão recebendo uma cópia de uma variável do tipo "Data" como parâmetro. Por exemplo, a função **iniciar.**

Passando estruturas como argumento

```
int main()
{
    struct Data dtHoje;
    dtHoje = iniciar( dtHoje );
    //...
}
struct Data iniciar( struct Data dt )
{
    //...
    dt.m_ok = 0 ;
    return dt ;
}
```

Passando estruturas como argumento

- Na função **main** criamos a variável **dtHoje** do tipo **struct Data**. Em seguida chamamos a função **iniciar** passando a variável **dtHoje** como parâmetro.

- O que ocorrerá aí será uma **cópia** do conteúdo armazenado na memória que apelidamos de "**dtHoje**" para uma outra memória, pertencente à função **iniciar**, a qual apelidamos de "**dt**".

- E **dt** é uma variável estruturada, com vários campos (e certamente essa estrutura teria ainda mais campos para atender a todas as funcionalidades necessárias). Assim todos esses campos precisarão ser copiados de um lugar da memória para outro. E isto **reduz a performance**.

Passando estruturas como argumento

- Além disso, a função **iniciar**, por sua vez, deve **retornar** uma **cópia** de **dt**, para que a função que a chamou possa receber as alterações que ela fez nesse parâmetro.
- Por isso tivemos que fazer:

  dtHoje = iniciar (dtHoje);

Passando estruturas como argumento

- Pois, do contrário as alterações seriam feitas em **dt**, parâmetro da função e não em dtHoje.

- Assim, para que o trabalho não seja perdido, deve haver uma nova cópia, desta vez em sentido inverso: desse modo o conteúdo de **dt** deve ser copiado para a área de retorno da função (o que é feito pela diretiva **return**) e, finalmente, deve ser copiado dessa área de retorno para a variável **dtHoje,** em **main** (o que é feito pelo operador de atribuição [**=**]).

Passando estruturas como argumento

- Poderíamos resolver tal problema com passagem de parâmetros por endereço ou referência.

Passando estruturas como argumento

- **Exemplo com passagem de parametros por endereço.**

```
void iniciar( struct Data * pdt )
{
    // ...
    *pdt.m_ok = 0 ;
}
```

Aqui existe um pequeno erro.

Passando estruturas como argumento

- Exemplo com passagem de parametros por endereço.
- 1º: Forma de corrigir

```
void iniciar( struct Data * pdt )
{
    // ...
    (*pdt).m_ok = 0 ;
}
```

Passando estruturas como argumento

- Exemplo com passagem de parametros por endereço.
- 2º: Forma de corrigir

```
void iniciar( struct Data * pdt )
{
    // ...
    pdt->m_ok = 0 ;
}
```

Passando estruturas como argumento

- Exemplo com passagem de parametros por referência.

```
void iniciar( struct Data &pdt )
{
    // ...
    pdt.m_ok = 0 ;
}
```

A especificação const

- Quando passamos um endereço ou uma referência, a função que o recebe passa a ter acesso a uma variável declarada em outra função, podendo alterá-la. Se a função que recebe endereços ou referências tiver como objetivo justamente o fornecimento de retornos extras, então é exatamente isso que se deseja.

A especificação const

- Mas se o objetivo for apenas velocidade de cópia(performance), então devemos deixar claro que essa função não irá alterar a memória cujo endereço ou referência ela recebeu e que usará esse endereço ou referência apenas para fins de leitura da variável apontada.

- Assim poderemos ter certeza que, caso essa memória passe a apresentar comportamentos indevidos, a origem do problema poderá estar em qualquer lugar, menos na função que recebeu o endereço ou a referência exclusivamente para leitura.

A especificação const

- Fazemos isso especificando ponteiros e referências em situações assim como const, o qual estabelece uma condição read-only para os dados. Nos exemplos acima, esse seria o caso da função data_imprime. Ela não foi projetada para alterar nada, e sim apenas para ler e imprimir. Então, ela deveria ser declarada de uma das duas maneiras abaixo:

- Usando ponteiros:

```
void imprimir( const Data * pdt ) ; // Os dados apontados são
      //'read-only' nesta função
```

- Usando referências:

```
void imprimir( const Data & dt ) ; // Os dados referenciados são
//'read-only' nesta função
```

Entrando definitivamente em oo

- Propriedades:
  - imaginem que tenhamos um objeto celular. esse objeto possui suas propriedades que nada mais são que seus atributos como, por exemplo: alto-falante, antena, microfone, memória, processador, câmera, etc...
  - Esses itens são atributos (Propriedades) do nosso objeto celular.
  - Obviamente que esses atributos (Propriedades) não foram feitos para serem acessados diretamente pelo usuário, eles são internos do celular e somente o mesmo poderá manipula-los diretamente e não nós.

Entrando definitivamente em oo

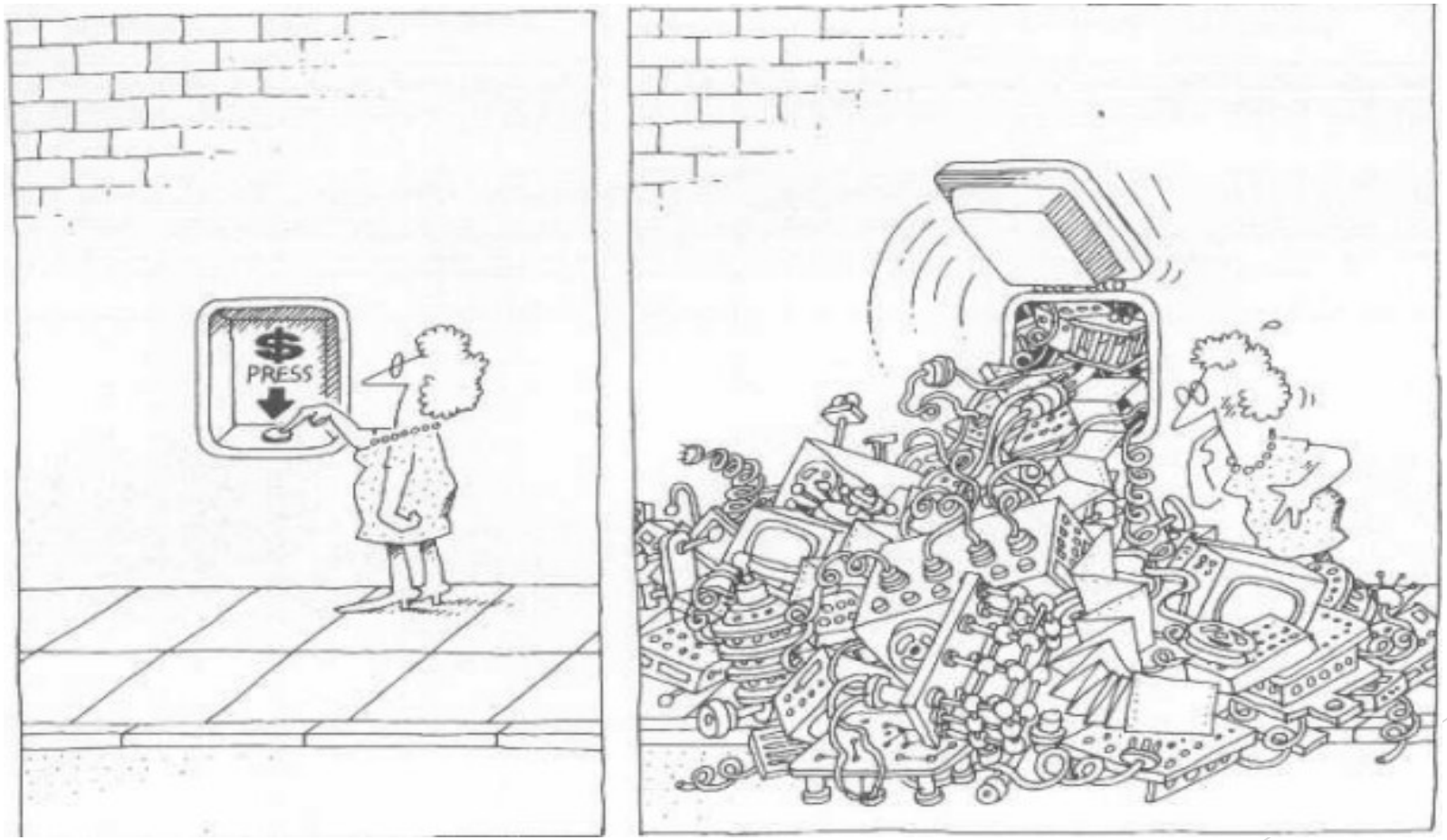
- Encapsulamento:
  - Imaginem que desejamos abaixar o volume do celular, nós não iremos acessar o alto-falante diretamente e realizar uma série de procedimentos para abaixar o volume diretamente.
  - Nesse caso faz sentido que o celular nos forneça dois botões, um aumenta o volume e o outro abaixa.
  - Quando eu aperto o botão de diminuir o volume, não interessa se foi enviado um sinal elétrico e o mesmo foi capturado por algum tipo de sistema e esse sistema interpretou o esse sinal e mandou a ordem para o alto-falante abaixar o volume. Isso é irrelevante para nós. Tudo o que interessa é saber que existem esse botões e saber que eles podem abaixar ou aumentar o volume.
  - Isso se chama encapsulamento, você sabe que existe algo, sabe a funcionalidade, mas os detalhes internos não lhe interessam.

Entrando definitivamente em oo

Ou seja, o encapsulamento deve passar uma “ilusão” de simplicidade.

Entrando definitivamente em oo

Ou seja, o encapsulamento deve passar uma “ilusão” de simplicidade.



Classe vs objeto

- Uma classe é um “molde” um “projeto”.
- Pense da seguinte forma, imaginem que fossemos construir um prédio.
- Inicialmente teríamos que ter:
  - Um projeto
  - E esse projeto iria especificar a altura do prédio.
  - Numero de quartos.
  - Cozinhas.
  - Banheiros.
  - E assim por diante.
- Ou seja, teríamos um molde completo de como o nosso prédio deve ser **exatamente.**
- Nesse caso dizemos que esse projeto do prédio seria a **classe predio.**

Classe vs objeto

Classe

Classe vs objeto

- Um objeto é uma instancia de uma classe.
- Pense da seguinte forma, imaginem que já temos o projeto do prédio pronto, especificando o que cada cômodo do prédio deve ter.
- Em seguida, contratamos uma equipe de pedreiros que vão erguer esse prédio com base no projeto.
- Assim que o projeto for completamente **CONSTRUÍDO** teremos um **OBJETO** do projeto prédio, ou como diríamos em OO: uma instância da classe prédio.
- Um objeto nada mais é do que uma instância de uma classe.
- Podemos ter várias instâncias de uma mesma classe, como no caso do projeto, o projeto desse prédio poderia ser construído (instanciado) em SP e depois poderíamos construir outros prédios exatamente iguais a esse em outras cidades como RJ, BH... Usando como "molde" o projeto.

Classe vs objeto

- Uma classe é um “molde” um “projeto”.
- Pense da seguinte forma, imaginem que fossemos construir um prédio.
- Inicialmente teríamos que ter:
  - Um projeto
  - E esse projeto iria especificar a altura do prédio.
  - Numero de quartos.
  - Cozinhas.
  - Banheiros.
  - E assim por diante.
- Ou seja, teríamos um molde completo de como o nosso prédio deve ser **exatamente.**
- Nesse caso dizemos que esse projeto do prédio seria a **classe predio.**

Classe vs objeto

Objeto

Estruturas em C++ : encapsulamento

- Vimos que em C não há como garantir o encapsulamento de um modo simples. Já em C++ isso será possível.

- Em C++ podemos declarar tanto membros de dados(campos) como também as próprias funções dentro da declaração da struct, de modo que tanto os campos de dados como funções sejam membros da estrutura.

```
struct Data
{
    int m_dia;
    int m_mes;
    int m_ano;
    void imprimir(const Data &dt)
    {
    //...
    }
    void alterar(Data &dt, int dia, int mes, int ano)
    {
        //...
    }
    void iniciar(Data &Dt)
    {
    //...
    }
};
```

Agora vamos implementar a nossa estrutura

```
struct Data {
    int m_dia;
    int m_mes;
    int m_ano;
    int m_ok;
    void alterar(Data &dt, int dia, int mes, int ano) {
        if( (dia >= 1   && dia <= 31)   &&  (mes>= 1 && mes <= 12 ) &&  (ano>= 1900 && ano<= 2100)) {
            dt.m_dia = dia;
            dt.m_mes = mes.
            dt.m_ano = ano;
            dt.m_ok= 1;
        }
        else
            dt.m_ok = 0;
    }
    void imprimir(const Data &dt) {
        if(dt.m_ok)
            cout << dt.m_dia << '/' << dt.m_mes << '/' << dt.m_ano;
         else
            cout << "Data Invalida.\n";
    }
    void iniciar(Data &dt) {
        dt.m_ok = 0;
    }
};
```

Ainda existem problemas

1°: Dia tem o valor máximo de 31, mês tem o valor máximo de 12 e ano máximo de 2100, será que o int é realmente o mais adequado?

2°: Estamos usando int para indicar se a data é correta ou incorreta, será que o int é o mais adequado para isso?

Ainda existem problemas

3°: O programador ainda pode acessar nossos membros de dados diretamente como, por exemplo:

```
int main()
{
    Data Dt;
    Dt.m_ok = 0;
    Dt.imprime(Dt);
    return 0;
}
```

4°: Quem garante que o programador sempre vai chamar a função inicia primeiro?

Solucionando o problema 1

- Sabemos que um dia não é menor que 1 e nem maior que 31, e que um mês não é menor que 1 e nem maior que 12 e um ano não poderá ser menor que 1900 e nem maior que 2100.

- Sabemos também que um int pode armazenar um valor superior a 4 bilhões e que ocupa 4 bytes na memória.

- Além disso também sabemos que um short armazena um valor de cerca de 65 mil, ou seja, um short consegue armazenar perfeitamente um ano ou mês ou dia.

Solucionando o problema 1

```
struct Data {
        short m_dia;
        short m_mes;
        short m_ano;
        int m_ok;
        void  alterar(Data &dt, int dia, int mes, int ano) {
                if( (dia >= 1   && dia <= 31)   &&  (mes>= 1 && mes <= 12 ) &&  (ano>= 1900 && ano<= 2100)) {
                        dt. m_dia = dia;
                        dt. m_mes = mes.
                        dt. m_ano = ano;
                        dt. m_ok= 1;
                }
                else
                        dt. m_ok = 0;
        }
        void imprimir(const Data &dt) {
               if(dt.m_ok)
                  cout << dt.m_dia << '/' << dt.m_mes << '/' << dt.m_ano;
                else
                   cout << "Data Invalida.\n";
        }
        void iniciar(Data &dt) {
               dt.m_ok = 0;
        }
};
```

Solucionando o problema 2

- Na nossa struct data temos um flag chamado ok que indica se há uma data válida ou não.
- Para isso apenas 2 valores nos bastariam, ou verdadeiro ou falso, entretanto usamos um int.
- Isso é um grande desperdício de memória, pois como estamos em C++ poderíamos usar apenas um bool para indicar se a data é válida ou não.

Solucionando o problema 2

```
struct Data {
    short m_dia;
    short m_mes;
    short m_ano;
    bool m_ok;
    void alterar(Data &dt, int dia, int mes, int ano) {
        if( (dia >= 1   && dia <= 31)   &&  (mes>= 1 && mes <= 12 ) &&  (ano>= 1900 && ano<= 2100)) {
            dt.m_dia = dia;
            dt.m_mes = mes.
            dt.m_ano = ano;
            dt.m_ok= true;
        }
        else
            dt.ok = false;
    }
    void imprimir(Data &dt) {
        if(dt.m_ok)
            cout << dt.m_dia << '/' << dt.m_mes << '/' << dt.m_ano;
        else
            cout << "Data Invalida.\n";
    }
    void iniciar(Data &dt) {
        dt.m_ok = 0;
    }
};
```

Solucionando o problema 3

- Os programadores podem acessar diretamente os membros de dados da nossa struct o que poderia ser trágico dependendo da situação.

- Para solucionar tal problema, as linguagens de programação orientadas a objeto em geral fornecem recursos para modificar o acesso entre esses dados.

- Dois deles que veremos agora são o private e o public.

Solucionando o problema 3

```
struct Data {
private:
    short m_dia;
    short m_mes;
    short m_ano;
    bool m_ok;
public:
    void alterar(Data &dt, int dia, int mes, int ano) {
        if( (dia >= 1   && dia <= 31)   &&  (mes>= 1 && mes <= 12 ) &&  (ano>= 1900 && ano<= 2100)) {
            dt.m_dia = dia;
            dt.m_mes = mês;
            dt.m_ano = ano;
            dt.m_ok= true;
        }
        else
            dt.m_ok = false;
    }
     void imprimir(Data &dt) {
        if(dt.m_ok)
           cout << dt.m_dia << '/' << dt.m_mes << '/' << dt.m_ano;
         else
            cout << "Data Invalida.\n";
    }
    void iniciar(Data &dt) {
        dt.m_ok = 0;
    }
};
```

Solucionando o problema 4

- Para que possamos resolver o problema 4 é necessário saber algumas coisas antes, como por exemplo:
- Já notaram que toda vez que chamamos uma função do nosso objeto passamos ele mesma como referência?

```
int main()
{
    Data Dt;
    Dt.imprime(Dt);
    return 0;
}
```

Solucionando o problema 4

- Será então que sempre seremos obrigados a passar o próprio objeto como parâmetro para que a função seja executada?

Solucionando o problema 4

- Na verdade... NÃO

- Em C++ sempre que existe uma chamada a uma função de uma classe, o compilador passa sempre um parâmetro oculto(exceto quando a função é static).

- Esse parâmetro se chama 'this'.

- 'this' é um ponteiro que aponta para quem chamou, logo quando fizemos:
  - Dt.imprime(/*this oculto*/, Dt);
- Vão existir dois parâmetros: 1º: o this oculto e o 2º: A referência para o próprio objeto que o chamou.

Solucionando o problema 4

- Ilustração do ponteiro this.
  - Debug mode:

main.cpp | std

```
2   #include <limits>
3
4   using namespace std;
5
6   struct Data
7   {
8   private:
9       short m_dia;
10      short m_mes;
11      short m_ano;
12      bool  m_ok;
13  public:
14      void alterarData(Data &dt, int dia, int mes, int ano)
15      {
16          if( (dia >= 1   && dia <= 31)    &&
17              (mes>= 1 && mes <= 12 )      &&
18              (ano>= 1900 && ano<= 2100))
19          {
20              dt.m_dia = dia;
21              dt.m_mes = mes;
22              dt.m_ano = ano;
23              dt.m_ok  = true;
24          }
25          else
26              dt.m_ok = false;
27      }
```

Name	Value	Type
ano	1991	int
dia	11	int
dt	@0x23fea8	Data &
m_ano	9326	short
m_dia	-116	short
m_mes	35	short
m_ok	64	bool
mes	8	int
this	@0x23fea8	Data
m_ano	9326	short
m_dia	-116	short
m_mes	35	short
m_ok	64	bool

Solucionando o problema 4

- Com isso ficou claro que o this possui o mesmo endereço de memória e os mesmos valores que o parâmetro dt que passamos por referência, logo percebemos que não precisamos passar essa referência e sim usar o ponteiro this.

Solucionando o problema 4

- Mas como se faz isso?

Solucionando o problema 4

```
void alterar(int dia, int mes, int ano)
{
    if( (dia >= 1   && dia <= 31)   &&
        (mes>= 1 && mes <= 12 )     &&
        (ano>= 1900 && ano<= 2100))
    {
        this->m_dia = dia;
        this->m_mes = mes;
        this->m_ano = ano;
        this->m_ok  = true;
    }
    else
        this->m_ok = false;
 }
//chamada a função ficaria assim
dtHoje.alterar(11, 8, 1991);
```

Solucionando o problema 4

- Entretanto o compilador coloca o ponteiro this implicitamente para nós, eliminando a necessidade de coloca-lo explicitamente.

```
void alterar(int dia, int mes, int ano)
  {
     if( (dia >= 1   && dia <= 31)   &&
         (mes>= 1 && mes <= 12 )     &&
         (ano>= 1900 && ano<= 2100))
     {
         m_dia = dia;
         m_mes = mes;
         m_ano = ano;
         m_ok  = true;
     }
     else
         m_ok = false;
  }
//chamada a função ficaria assim
dtHoje.alterar(11, 8, 1991);
```

Solucionando o problema 4

- Agora que descobrimos a funcionalidade do ponteiro this, finalmente poderemos resolver o problema 4.
- Para toda nova classe que se for criar existe uma "coisa" chamada função construtora ou método construtor.
- Essa função é uma função especial e possui uma sintaxe especial, tendo o mesmo nome da classe, seguido por abre e fecha parênteses e depois abre e fecha chaves.

```
CLASSE()
{
}
```

Solucionando o problema 4

- A função construtora é a primeira que será chamada logo quando se instancia um objeto da nossa classe.
- Sendo assim ela poderia iniciar os seus membros de dados com um valor default e com isso fechamos “parcialmente” os problemas da data.

```
struct Data {
private:
    short m_dia;
    short m_mes;
    short m_ano;
    bool  m_ok;
public:
    Data() {
        m_ok = 0;
    }
};
```

Vantagens

1) Em C++ as funções fazem parte da estrutura, como membros.

2) E o acesso aos dados deverá ser feito obrigatoriamente através de funções declaradas dentro da struct caso eles tenham sido declarados como privativos da estrutura (private).

3) No momento da declaração da variável (inicialização) é chamada automaticamente a função que tem o mesmo nome da estrutura (construtora).

4) Não precisamos (e não devemos) passar como parâmetro a variável estruturada ou o seu endereço: isto será feito automaticamente pelo compilador (pois o compilador acrescentará um parâmetro oculto contendo o endereço da variável estruturada que disparou a chamada à função). E o nome convencionado para esse parâmetro oculto é "this"(que, em princípio, deve ser passado por registrador e não pela pilha).

Exercício

- Criar uma struct(classe) que permita armazenar dados de clientes.
- Esses dados serão: Nome, sobrenome, sexo e endereço.
- Esses membros de dados não devem ser acessados diretamente, ou seja, criem uma função membro para acessar os dados desse cliente e outra para imprimir o cliente.

Tempo 20 minutos

Exercício

- Sabemos que uma empresa não possui somente um cliente.
- Modifiquem o programa de modo que o usuário possua 3 opções.
  - 1º: Cadastrar novo cliente.
  - 2º: Imprimir lista de clientes cadastrados
  - 3º: Sair

- OBS: Usem a classe vector para isso.

Tempo 30 minutos