

# Lista de exercícios

Ex. 1)

## CONSTRUTORES:

### *O que são?*

Construtores, como o nome já diz, irão construir algo. No caso, o construtor deste exercício irá "construir" os valores padrões de cada classe, ou seja, elas começarão com um valor predefinido.

```
class Data {  
    private:  
        int dia, mes, ano;          // <- Atributos  
  
    public:  
        ...  
  
        Data() : dia(22), mes(2), ano(2005) {}; // Construtor PADRÃO  
  
        ...  
  
};
```

### *Construtor PARAMETRIZADO:*

Todo método pode receber um parâmetro. No caso do construtor não é diferente, por se tratar de um método padrão da classe, ele pode receber parâmetros.

Nesse caso, os parâmetros serão os valores por padrão que serão inseridos no objeto (dia, mês e ano).

```
class Data {  
    ...  
  
    public:  
        Data(int, int, int);          // <- Os parâmetros são inseridos dentro dos  
                                       // parênteses. No caso, os parâmetros são os  
                                       // "Tipos" de parâmetros que o método vai receber  
                                       // (do tipo INTEIRO).  
  
    ...  
};
```

```

...

Data::Data(int dia, int mes, int ano) {
    this->dia = dia;
    this->mes = mes;
    this->ano = ano;
}

// Já no método em si, os parâmetros precisam ser declarados novamente. No caso o tipo
// e o nome do valor (não necessariamente precisa ser “dia” ou “mes”). No exemplo acima
// os parâmetros “dia”, “mês” e “ano” NÃO se tratam dos atributos da classe, mas sim
// dos VALORES QUE SERÃO RECEBIDOS. Por conta disso eles podem ter qualquer nome, pois
// serve apenas como um indicador de qual é o nome valor que está sendo recebido

...

```

Sobre o código acima, o que é o “this”?

No exemplo acima, como estamos utilizando o mesmo nome para definir o valor do parâmetro e do atributo da classe, precisamos utilizar o “this” para dizer:

***“Este->dia se refere ao ATRIBUTO da CLASSE, não ao parâmetro dia”.***

Ai você me diz: “Krai, que trampo fazer isso tudo, é só colocar um nome diferente pro parâmetro kkkkkkkj”. Sim, podemos fazer isso, porém se trata de uma boa prática da programação, logo, utilizamos o mesmo nome tanto para o parâmetro quanto ao atributo da classe.

Logo, a linha de pensamento para elaboração de um CONSTRUTOR PARAMETRIZADO é essa abaixo:

```

class Data {
    private:
        int dia, mes, ano; // <- Atributos
        ...
};

Data::Data(int dia, int mes, int ano) {    // Os atributos está dentro dos parênteses
    this->ATRIBUTO = PARÂMETRO;
    this->ATRIBUTO = PARÂMETRO;
    this->ATRIBUTO = PARÂMETRO;
}

```

## Como definir os valores padrões dentro da main() ?

```
#include <iostream>

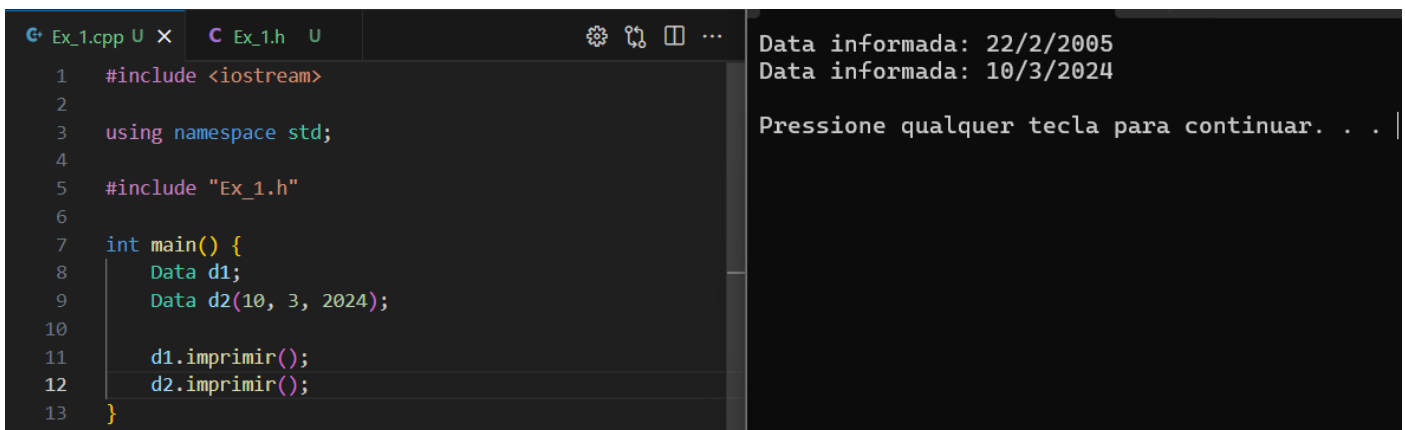
using namespace std;

#include "Ex_1.h"

int main() {
    Data d1;                // <- O valor será do construtor padrão (22, 2, 2005)
    Data d2(10, 3, 2024);   // <- O valor será definido de acordo com os parâmetros
}
```

Após criarmos o objeto “d1”, logo em seguida abrimos o parêntese e já inserimos os valores padrões do tipo inteiro.

## Execução:



```
Ex_1.cpp U x  C Ex_1.h U
1  #include <iostream>
2
3  using namespace std;
4
5  #include "Ex_1.h"
6
7  int main() {
8      Data d1;
9      Data d2(10, 3, 2024);
10
11     d1.imprimir();
12     d2.imprimir();
13 }
```

```
Data informada: 22/2/2005
Data informada: 10/3/2024

Pressione qualquer tecla para continuar. . . |
```

## DESTRUTORES:

### O que são?

Os destrutores são métodos que LITERALMENTE destroem um objeto. Ou seja, ele é excluído da memória.

```
class Data {
    private:
        int dia, mes, ano;

    public:
        ...

        // DESTRUTOR
        ~Data() { cout << "O OBJETO FOI DESTRUÍDO" << endl; };

        ...
};
```

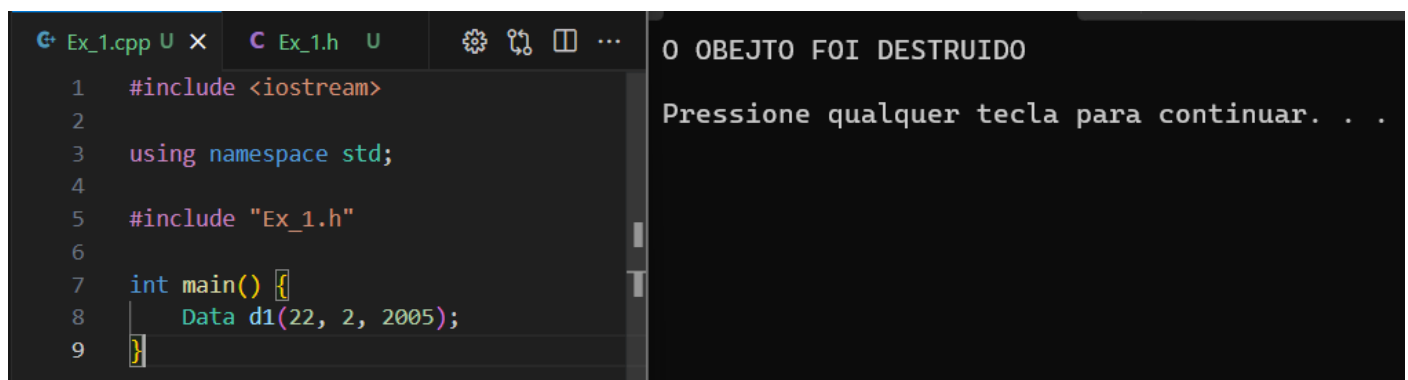
No exemplo acima, o destrutor é chamado juntamente com uma função *inline*, ou seja, uma função em linha. Nesse caso, função inline deste destrutor apenas exibirá uma mensagem após ser destruída.

### Declaração:

```
~NOME_DA_CLASSE() {}
```

### Como executar o destrutor?

O destrutor é executado AUTOMATICAMENTE ao final da chamada do objeto, ou seja, no final da execução de todos os métodos etc.

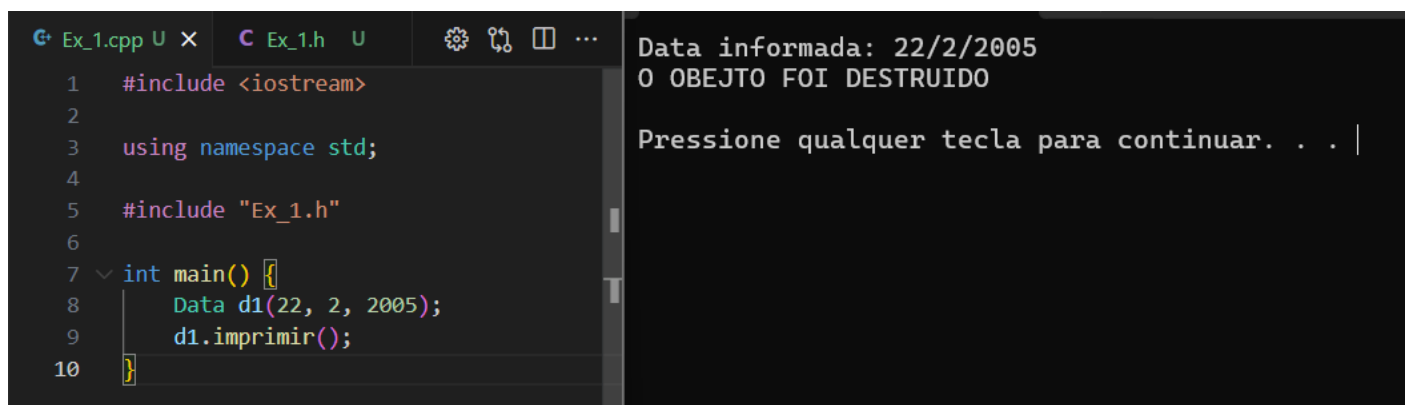


```
Ex_1.cpp U x C Ex_1.h U
1  #include <iostream>
2
3  using namespace std;
4
5  #include "Ex_1.h"
6
7  int main() {
8      Data d1(22, 2, 2005);
9  }
```

O OBJETO FOI DESTRUÍDO

Pressione qualquer tecla para continuar. . .

No exemplo acima, como apenas estanciamos o objeto da classe “Data” e não declaramos nenhum método, ele será executado no final do código.



```
Ex_1.cpp U x C Ex_1.h U
1  #include <iostream>
2
3  using namespace std;
4
5  #include "Ex_1.h"
6
7  int main() {
8      Data d1(22, 2, 2005);
9      d1.imprimir();
10 }
```

Data informada: 22/2/2005

O OBJETO FOI DESTRUÍDO

Pressione qualquer tecla para continuar. . . |

Já no exemplo acima, como executamos um método, o Destrutor será executado após a execução do método.

## LER():

O método ler() será utilizado para o USUÁRIO inserir o valor do objeto.

```
class Data {  
    private:  
        int dia, mes, ano;  
  
    public:  
        ...  
        void ler();  
        ...  
};
```

```
void Data::ler() {  
    cout << "Informe o dia: " << endl;  
    cin >> dia;  
    cout << "Informe o mes: " << endl;  
    cin >> mes;  
    cout << "Informe o ano: " << endl;  
    cin >> ano;  
    cout << endl;  
}
```

### *Declaração:*

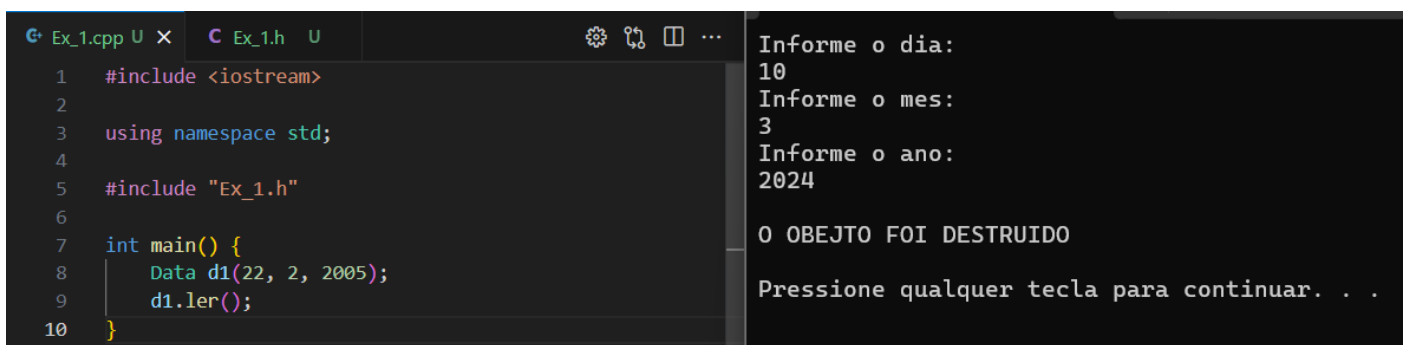
```
Tipo_do_método NOME_DA_CLASSE::NOME_DO_MÉTODO() {  
    ...  
    ...  
}
```

Tipo do método: int (retorna um valor) ou void (não retorna nada);

### *Como ficará na main()?*

```
#include <iostream>  
  
using namespace std;  
  
#include "Ex_1.h"  
  
int main() {  
    Data d1(22, 2, 2005);  
    d1.ler();  
}
```

### *Execução:*



```
Ex_1.cpp U x  C Ex_1.h U  [Icons]  
1  #include <iostream>  
2  
3  using namespace std;  
4  
5  #include "Ex_1.h"  
6  
7  int main() {  
8      Data d1(22, 2, 2005);  
9      d1.ler();  
10 }
```

```
Informe o dia:  
10  
Informe o mes:  
3  
Informe o ano:  
2024  
  
O OBJETO FOI DESTRUIDO  
  
Pressione qualquer tecla para continuar. . .
```

## IMPRIMIR():

O método imprimir() será utilizado para mostrar os valores na tela.

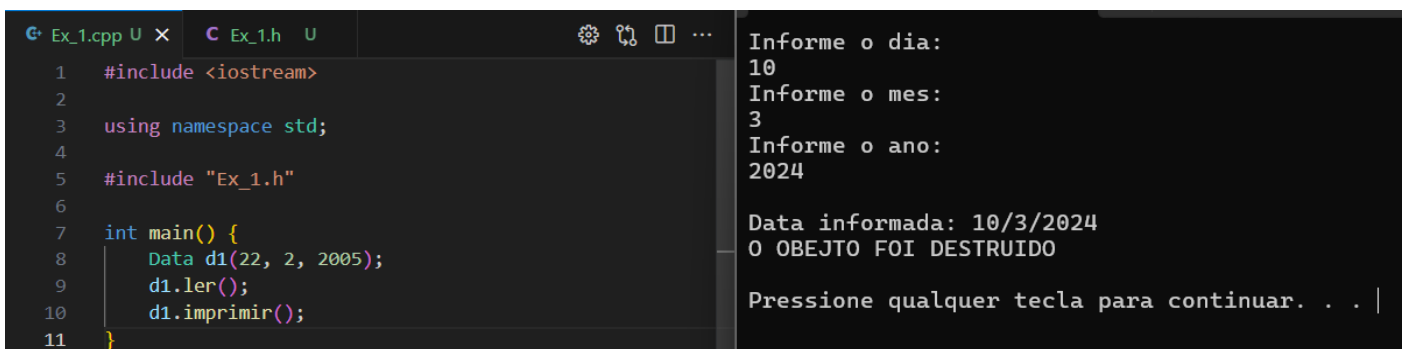
```
class Data {  
    private:  
        int dia, mes, ano;  
  
    public:  
        ...  
  
        void imprimir();  
        ...  
};
```

```
void Data::imprimir() {  
    cout << "Data informada: "  
    << dia << "/" << mes << "/" << ano << endl;  
}
```

### Como ficará na main()?

```
#include <iostream>  
  
using namespace std;  
  
#include "Ex_1.h"  
  
int main() {  
    Data d1(22, 2, 2005);  
    d1.ler();  
    d1.imprimir();  
}
```

### Execução:



```
Ex_1.cpp U x C Ex_1.h U  
1  #include <iostream>  
2  
3  using namespace std;  
4  
5  #include "Ex_1.h"  
6  
7  int main() {  
8      Data d1(22, 2, 2005);  
9      d1.ler();  
10     d1.imprimir();  
11 }
```

```
Informe o dia:  
10  
Informe o mes:  
3  
Informe o ano:  
2024  
  
Data informada: 10/3/2024  
O OBJETO FOI DESTRUIDO  
  
Pressione qualquer tecla para continuar. . . |
```

## GET's & SET's, a importância de utilizá-los:

Para entendermos o conceito de GET e SET, devemos lembrar de um dos “Mantras” da programação orientada à objeto, o **ENCAPSULAMENTO**.

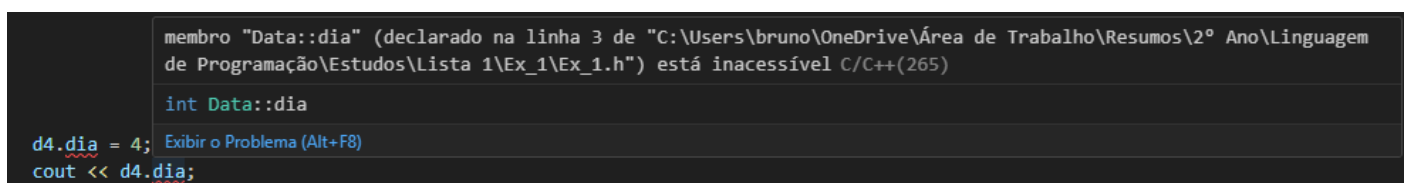
Resumidamente, o conceito de encapsulamento diz que “Os atributos do objetos não podem ser manipulados diretamente em uma função que não estejam diretamente ligada à classe”, ou seja, a main() é um exemplo disso.

Não podemos alterar diretamente o atributo de um objeto acessando o mesmo, como no exemplo abaixo:

```
int main() {  
    ...  
  
    d4.dia = 4;  
    cout << d4.dia;  
  
    ...  
}
```

Este exemplo fere vai totalmente contra o conceito de programação orientada à objeto, pois, como pode ser observado, está sendo alterado diretamente os atributos do objeto e exibindo ele logo em seguida

É importante seguirmos essa regra pra termos mais segurança em nosso código e evitarmos possíveis erro dos usuários. Por conta disso deixamos os atributos como “Private” e os métodos como “Public”. Assim, os acessos às classes não poderão ocorrer diretamente na main() ou em outras funções que não estejam relacionadas diretamente à classe.



*Erro ao tentar manipular um objeto que está privado. Ele se torna inacessível*

Para podermos acessar os valores e manipulá-los, devemos ter um “intermediário” para isso. Logo, o “get” e o “set” serão nossos intermediários.

## GET's:

*Para que serve?*

Como seu nome já diz, ele “pega” o valor, o qual pode ser usado da maneira que quiser, seja o exibindo, o somando a outro número... você pode utilizá-lo da maneira que achar melhor

O “get” também é um método que retorna um valor, ou seja, ele será uma função do tipo int.

*Neste caso ele está retornando uma data, ou seja, um número, logo a função será do tipo INT pois retorna um número. Caso retornasse uma palavra, seria uma função do tipo STRING, se fosse um número com vírgula seria uma função do tipo float etc.*

### Declaração:

```
class Data {
    private:
        int dia, mes, ano;

    public:
        ...
        int getDia() {return dia;};      // <- 'get', função inline para obter o dia
        int getMes() {return mes;};     // <- = mes
        int getAno() {return ano;};     // <- = ano
        ...
};
```

### Exemplo:

```
int main() {
    ...

    Data d5(10, 2, 2005);
    cout << "Dia do objeto D5 + 10: " <<
    endl << "Valor pr3 setado: " << d5.getDia() <<
    endl << "Valor do dia + 10: " << (d5.getDia() + 10) <<
    endl << endl << endl;

    ...
}
```

No exemplo acima, primeiramente, estanciamos o objeto d5 para classe Data. Depois, por meio do construtor parametrizado, incluímos os valores dos atributos do objeto d5.

Após incluído, utilizamos o método getData() para exibir o valor na tela e depois somar o valor dele por 10.

*Após somarmos, o valor do dia não foi alterado, apenas somamos mais 10. No caso, o valor de dia continua sendo 10, e não 20.*



## SET's:

### *Para que serve?*

Como seu nome já diz, o “set” vai definir um valor ao objeto.

Ele também se trata de um método, logo, também será declarado no escopo da classe “Data”.

O “set” não retorna nenhum valor, portanto ele se trata de uma função do tipo void.

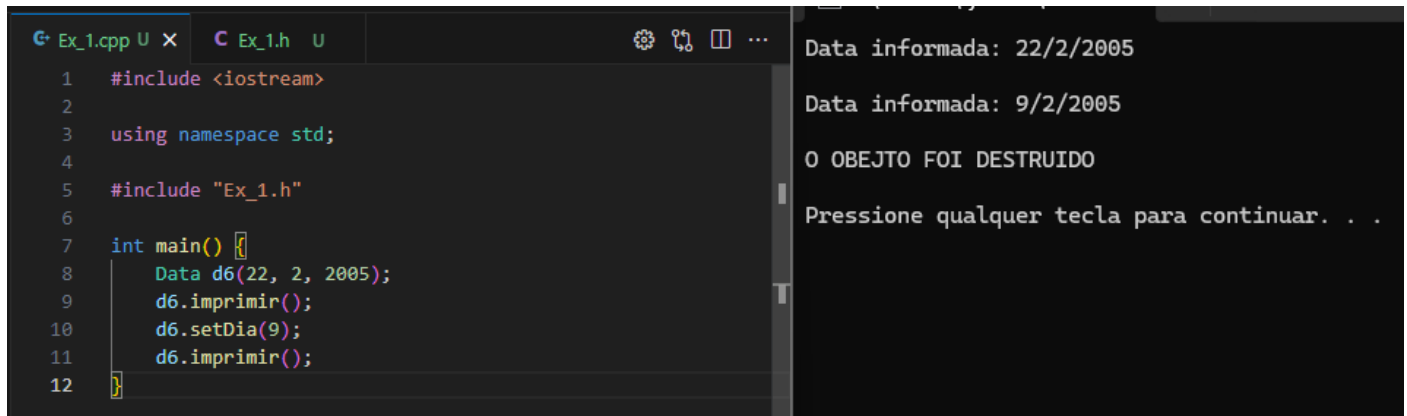
*\*Obs: no exercício, é pedido que sejam adicionadas regras para inclusão de datas. Exemplo: dia não pode ser mais que 31 ou 30, dependendo do mês. Não irei incluir as regras (que são if's para exceções), apenas irei mostrar o funcionamento dele.*

*As exceções serão incluídas no código final.*

### *Declaração:*

```
class Data {  
    private:  
        int dia, mes, ano;  
  
    public:  
        ...  
        int setDia(int);  
        int setMes(int);  
        int setAno(int);  
        ...  
};  
  
void Data::setDia(int dia) {  
    this->dia = dia;  
}  
  
void Data::setMes(int mes) {  
    this->mes = mes;  
}  
  
void Data::setAno(int ano) {  
    this->ano = ano;  
}
```

### Exemplo:



The screenshot shows a C++ IDE with two panels. The left panel displays the source code for `Ex_1.cpp` and `Ex_1.h`. The right panel shows the program's output.

```
1  #include <iostream>
2
3  using namespace std;
4
5  #include "Ex_1.h"
6
7  int main() {
8      Data d6(22, 2, 2005);
9      d6.imprimir();
10     d6.setDia(9);
11     d6.imprimir();
12 }
```

Output:

```
Data informada: 22/2/2005
Data informada: 9/2/2005
O OBJETO FOI DESTRUIDO
Pressione qualquer tecla para continuar. . .
```

No exemplo acima, estanciamos o objeto `d6` à classe `Data`. Depois, por meio do construtor parametrizado incluímos os valores aos atributos.

Quando imprimimos pela primeira vez, é mostrado a data inserida pelo construtor. Depois de utilizamos o método `"setDia"` para alterar o valor do dia para 9.

Após chamarmos novamente o método `imprimir()`, é exibido o valor atualizado do objeto.