

# CLASSES - Regras gerais

Tipos de membros das classes:

- Dados membro: similares aos existentes nas `struct`.
- Funções membro (também denominados métodos).

Comunicação de funções membro com quem a chamou:

- Parâmetros de entrada (iguais às das funções normais).
- Parâmetro de retorno (igual às das funções normais).
- Objeto sobre o qual a função membro atua.
  - Só existe nas funções membro
  - Pode ser visto, no escopo da função membro, como um dado global compartilhado entre todas as funções membro da mesma classe, quando atuarem no mesmo objeto.

Divisão público x privado:

- Dados membro geralmente são privados.
- Funções membro geralmente são públicas.

Construtores e destrutores

- Construtores e destrutores são métodos especiais que sempre são chamados quando uma variável da classe é criada.
  - Pode haver mais de um construtor, de acordo com o tipo de parâmetro
  - Só pode haver um destrutor.
- Os construtores de uma classe devem utilizar e referenciar os construtores apropriados das classes dos objetos dos quais a classe é composta:

```
class Base                                class Deriv
{
    int P1;
    double P2;
public:
    Base(int I):
        P1(I), P2(0.0)
    { ... }
};
                                        {
    Base B;
    int P3;
public:
    Deriv(int I, int J):
        B(I), P3(J)
    { ... }
};
```

- Sempre existirá um construtor por cópia (parâmetro é outra variável da mesma classe):
  - Se você não fizer um, o compilador criará um (copiando todos os valores byte a byte - isso não é o que você deseja caso a classe tenha algum dado do tipo ponteiro).
  - O construtor por cópia sempre recebe como único parâmetro um objeto (geralmente `const`) da classe em questão, SEMPRE passado por referência (com "&"):  
`Classe(const Classe &X)`
- Quase sempre existirá um construtor default (nenhum parâmetro):
  - Se você não fizer nenhum construtor, o compilador criará um default (deixa todos os dados com lixo, o que não é admissível caso a classe tenha dado do tipo ponteiro).
  - Caso você crie algum construtor, o compilador não criará o construtor default.
- Toda classe que utilizar alocação dinâmica de memória, tem OBRIGATORIAMENTE que prever:
  - Construtor default
  - Construtor por cópia
  - Destrutor
  - Operador de atribuição (`operator=`) - Ver abaixo

Funções de consulta

- Prever as funções de consulta (ex.: `get____` ) e de fixação de valores (Ex.: `set____` ).
- Muitas vezes as funções `get____` podem ser inline.
- As funções `set____` (e algumas `get____` ) muitas vezes devem checar parâmetros.
- Muitas vezes as funções de consulta são definidas por sobrecarga dos operadores `()` ou `[]`.

#### Sobrecarga de operadores

- Se possível, os operadores devem ser sobrecarregados como funções membro (métodos):  
Exemplo `A+B`:
  - `operator+(A,B) -> função C (evitar).`
  - `A.operator+(B) -> função-membro C++ (preferir, quando possível).`
- Sobrecarregar os operadores apenas com o sentido usual (não usar operadores para outras coisas).
- Os operadores `>>` e `<<` devem ser funções friend, não métodos (funções membro):
 

```
friend ostream &operator<<(ostream &X, const ____ &M) ;
friend istream &operator>>(istream &X, ____ &M) ;
```
- Toda classe com alocação dinâmica de memória deve OBRIGATORIAMENTE sobrecarregar o `operator=`
  - O `operator=` geralmente é composto pelo código igual ao do destrutor (limpar o conteúdo anterior) seguido do código igual ao do construtor por cópia (fazer a cópia).

#### Funções auxiliares

- Uma estratégia usual para classes com alocação dinâmica de memória é sempre criar 2 funções, limpar e copiar, que podem ser privadas. Com essas 2 funções, você consegue criar 3 funções-membro obrigatórias sem repetir código:
  - Construtor por cópia = copiar
  - Destrutor = limpar
  - Operador de atribuição (`operator=`) = limpar + copiar

#### Grandezas constantes

- Ao definir funções e métodos, sempre indicar quando uma grandeza é constante:
 

```
const ____ metodo(const ____) const
```

  - O primeiro `const` indica que o valor de retorno não pode ser modificado por quem o receber (pouco usado).
  - O segundo `const` indica que o método não pode alterar o parâmetro (geralmente só qdo é passado com `&`).
  - O terceiro `const` indica que o método não pode alterar o objeto no qual é chamado.

#### Passagem de parâmetros

- Objetos "grandes" são passados por referência, mesmo quando não se pretende alterá-los (usar o `const` nesse caso):
  - `funcao(Classe_Grande X) -> ERRADO: faz uma cópia desnecessária de X`
  - `funcao(Classe_Grande &X) -> OK, quando precisa alterar X`
  - `funcao(const Classe_Grande &X) -> OK, quando não pode alterar X`

#### Utilização do construtor por cópia (3 situações):

- Ao criar uma nova variável passando como argumento outra variável do mesmo tipo:
 

```
Classe X;
Classe Y(X);    ou    Classe Y=X;
```
- Ao passar um parâmetro por cópia para uma função:
 

```
void funcao(Classe X)
{ ... }
```

```
Classe Z;  
funcao(Z); // X eh criada como sendo uma copia de Z
```

- **Ao retornar um objeto como valor de retorno de uma função:**

```
Classe funcao(void)  
{ Classe prov;  
  ...  
  return prov;}  
// Uma variável sem nome eh criada no programa principal como  
// copia de prov para conter o valor de retorno da funcao  
cout << funcao();
```

#### Utilização do construtor específico (2 situações):

- **Ao criar uma nova variável passando como argumento variável(is) de outro(s) tipo(s):**

```
Outra_Classe X;  
Classe Y(X); ou Classe Y=X;
```

- **Ao promover um objeto (apenas para construtor específico com um único parâmetro):**

```
class Classe { ...  
  Classe(int i); // Construtor a partir de um inteiro  
};
```

```
void funcao(Classe X, Classe Y)  
{ ... }
```

```
Classe M;
```

```
funcao(M,1); // Serah chamado construtor para converter o 1
```

- **Para impedir que um construtor específico seja utilizado em promoções, deve-se utilizar a palavra-chave explicit:**

```
explicit Classe(int i); // Nao eh usado em promocoies  
função(M,1); // Erro de compilacao
```