PROGRAMAÇÃO ORIENTADA A OBJETOS

Conceitos gerais:

- A Programação Orientada a Objetos (POO) é um paradigma de programação baseado no conceito de *objetos*, que podem conter tanto dados (campos, atributos ou propriedades) quanto código (procedimentos ou métodos).
  - Os objetos são como se fossem uma evolução do conceito de variáveis.
- Os procedimentos de um objeto podem acessar e geralmente modificar os campos de dados do objeto ao qual estão associados (objetos têm uma noção de si mesmos, `this`).
- Em POO, os programas são projetados criando-se objetos que interagem entre si.
- A maioria das linguagens POO, inclusive C++, baseia-se em *classes* para construir objetos.
  - As classes são como se fossem uma evolução do conceito de `struct`.

Principais características da POO:

- Classes: definições para o formato dos dados e os procedimentos disponíveis para um determinado tipo de objetos com características em comum.
- Objeto: instâncias de classes. Os objetos às vezes correspondem a coisas encontradas no mundo real. São uma expansão do conceito de variáveis.
  - Objetos fornecem uma camada de abstração que pode ser usada para separar o código interno (que integram a classe) do externo (de quem usa a classe).
- 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.
- Encapsulamento: conceito de POO que une os dados e as funções que manipulam esses dados, mantendo os dois protegidos contra interferências externas e uso indevido.
  - Permite a *ocultação de dados*: proteção de outras partes do programa de modificações se a estrutura interna do objeto for alterada. A proteção envolve o fornecimento de uma interface estável que oculta os detalhes com maior probabilidade de alteração.
- Construção incremental de objetos: objetos maiores podem ser projetados baseando-se em objetos menores pré-existentes.
  - *Composição*: objetos podem conter outros objetos em seus campos de dados. A composição representa relações do tipo "contém-um" ou “é-composto-de” entre objetos.
  - *Herança*: permite que as classes sejam organizadas em uma hierarquia que represente relações do tipo "é-um-tipo-de" entre objetos.
- Polimorfismo: interface única para entidades de diferentes tipos ou o uso de um único símbolo para representar vários tipos diferentes.
  - Polimorfismo de função (ou *ad hoc*): funções que podem ser usadas com argumentos de tipos diferentes, mas que se comportam de maneira diferente, dependendo do tipo de argumento ao qual elas são aplicadas. Conhecido como sobrecarga de função ou sobrecarga de operador em C++.
  - Polimorfismo paramétrico: permite que uma função ou um tipo de dados seja escrito genericamente, para que ele possa manipular valores de maneira uniforme sem depender de seu tipo. Implementado através dos `template` em C++.
  - Polimorfismo por subtipagem: quando o código de chamada de uma função é independente de qual classe na hierarquia de herança executará a função - a classe pai ou um de seus descendentes. Permite que uma função seja escrita para receber como parâmetro um objeto de um determinado tipo T, mas também funcione corretamente se for passado um objeto que pertence a um tipo S, que é um subtipo de T (herda de T). Em C++, é implementado através dos métodos virtuais.

CLASSES C++

Tipos de membros das classes:

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

Divisão público x privado:

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

Mecanismos de comunicação das funções membro (métodos):

- 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.
- Ao definir métodos, deve-se indicar quando uma dessas 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.
- Objetos "grandes" são geralmente 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

Métodos 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.

Construtores e destrutores

- Construtores e destrutores são métodos especiais que sempre são chamados quando uma variável da classe é criada ou deixa de existir.
  - `class MyClass` -> construtor denominado `MyClass`
  - `class MyClass` -> destrutor denominado `~MyClass`
  - 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 {
  int P1;
  double P2;
public:
  Base(int I):
    P1(I), P2(0.0)
  { … }    };
```

```
class Deriv {
  Base B;
  int P3;
public:
  Deriv(int I, int J):
    B(I),P3(J)
  { … }     };
```

- Alguns construtores recebem denominações próprias:
  - Construtor default: não tem nenhum parâmetro. É chamado quando o objeto é criado sem nenhuma inicialização.
    `MyClass A;`
  - Construtor por cópia: recebe um único parâmetro que é uma referência a outro objeto da mesma classe;
    `MyClass A;`
    `MyClass B(A);` ou `MyClass B=A;`
  - Todos os outros construtores são chamados de construtores específicos.
    `int a;`
    `MyClass A(a);` ou `MyClass A(a, 2.74);`
- Sempre existirá um construtor por cópia.
  - 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 "&"):
    `MyClass(const MyClass &X)`
- Quase sempre existirá um construtor default.
  - 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.
  - Classes sem construtor default não permitem a criação de objetos sem parâmetro (`MyClass X;`) nem a criação de arrays estáticos de objetos (`MyClass X[10];`), já que nessas horas seria chamado o construtor default (1 ou 10 vezes).
- Toda classe que utiliza alocação dinâmica de memória tem obrigatoriamente que prever:
  - Construtor default
  - Construtor por cópia
  - Destrutor
  - Operador de atribuição (`operator=`) - Ver abaixo

Amigas (`friend`) de uma classe

- As funções amigas de uma classe são funções clássicas que, apesar de não serem funções membro da classe, podem acessar os seus membros privados.

```
class MyClass { …
  friend myFunc(int i);  // myFunc nao eh membro de MyClass
};
```

- Se uma classe B é amiga da classe A, os membros da classe B podem acessar os membros privados da classe A.

```
class A { …
  friend class B; };
```

Sobrecarga de operadores

- C ++ permite usar operadores padrões da linguagem para executar operações não só com os tipos fundamentais, mas também com classes criadas.

```
                                  class MyClass { … };
int a,b,c;                        MyClass A,B,C;
…                                 …
a = b+c;                          A = B+C;
```

- Para sobrecarregar um operador e usá-lo com classes, declaramos *funções operador*, que são funções cujos nomes são a palavra-chave `operator` seguida pelo sinal do operador que queremos sobrecarregar.
  - tipo `operator` □`( parâmetros ) { … }`, onde □ é o símbolo do operador.

- Vários operadores podem ser sobrecarregados:
  ```
    +      -      *      /     =     <     >     +=    -=    *=    /=    <<    >>
   <<=    >>=    ==     !=    <=    >=    ++    --    %     &     ^     !     |
    ~      &=     ^=     |=    &&    ||    %=    []    ()    ,     ->*   ->    new
  delete  new[]  delete[]
  ```
- Os operadores devem ser sobrecarregados apenas com o sentido usual ou próximo.
- Para alguns operadores, existem duas maneiras de sobrecarregá-los: como uma função membro ou como uma função clássica. Seu uso é indistinto, no entanto, funções que não são membros de uma classe não podem acessar os membros privados ou protegidos dessa classe, a menos que a função seja `friend`.
  - Forma de declaração das funções operador, onde `a` é um objeto da classe `A`, `b` é um objeto da classe `B` e `c` é um objeto da classe `C`:

Expressão	Operador	Função membro	Função clássica
□a	`+ - * & ! ~ ++ --`	`A::operator□()`	`operator□(A)`
a□	`++ --`	`A::operator□(int)`	`operator□(A,int)`
a□b	`+ - * / % ^ & | < > == != <= >= << >> && || ,`	`A::operator□(B)`	`operator□(A,B)`
a□b	`= += -= *= /= %= ^= &= |= <<= >>= []`	`A::operator□(B)`	-
`a(b, c...)`	`()`	`A::operator()(B,C...)`	-
`a->x`	`->`	`A::operator->()`	-

  - Se possível, os operadores devem ser sobrecarregados como funções membro:
    `A.operator+(B)` -> função membro (preferir, quando possível).
    `operator+(A,B)` -> função clássica (evitar).
- Os operadores >> e << devem ser funções clássicas `friend`, não funções membro:
  ```
  friend ostream &operator<<(ostream &X, const ___  &M);
  friend istream &operator>>(istream &X, ___  &M);
  ```
- Toda classe com alocação dinâmica de memória precisa 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).

Utilização do construtor por cópia (3 situações):

- Ao criar uma nova variável passando como argumento outra variável do mesmo tipo:
  ```
  MyClass X;
  MyClass Y(X);  ou  MyClass Y=X;
  ```
- Ao passar um parâmetro por cópia para uma função:
  ```
  void myFunc(MyClass X) {  …  }
  MyClass Z;
  myFunc (Z);   // X eh criada como sendo uma copia de Z
  ```
- Ao retornar um objeto como valor de retorno de uma função:
  ```
  MyClass myFunc(void)
  { MyClass 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 << myFunc();
  ```

Utilização do construtor específico (2 situações):

- Ao criar uma nova variável passando como argumento variável(is) de outro(s) tipo(s):
  ```
  Outra_Classe X;
  MyClass Y(X);  ou  MyClass Y=X;
  ```
- Ao promover um objeto (apenas para construtor específico com um único parâmetro):
  ```
  class MyClass { …
    MyClass(int i);  // Construtor a partir de um inteiro
  };
  void myFunc(MyClass X, MyClass Y) {  …  }
  ```

```
MyClass M;
myFunc(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 MyClass(int i);   // Nao eh usado em promoções
myFunc(M,1);  // Erro de compilação
```

Funções auxiliares

- Uma estratégia útil para classes com alocação dinâmica de memória é criar 3 funções auxiliares, `criar`, `copiar` e `limpar`. Com essas 3 funções, você consegue criar algumas funções-membro obrigatórias sem repetir código:
  - `criar`: faz alocação dinâmica de memória (deve ser privada).
  - `copiar`: `criar` + cópia dos dados (deve ser privada).
  - `limpar`: liberação de memória (pode ser pública).
  - Construtor específico: geralmente usa `criar`.
  - Construtor por cópia = usa `copiar`.
  - Destrutor = usa `limpar`.
  - Operador de atribuição (`operator=`) = `limpar` + `copiar`.