

Aula 04:



Classes – Parte II

ECOP13A - Programação Orientada a Objetos

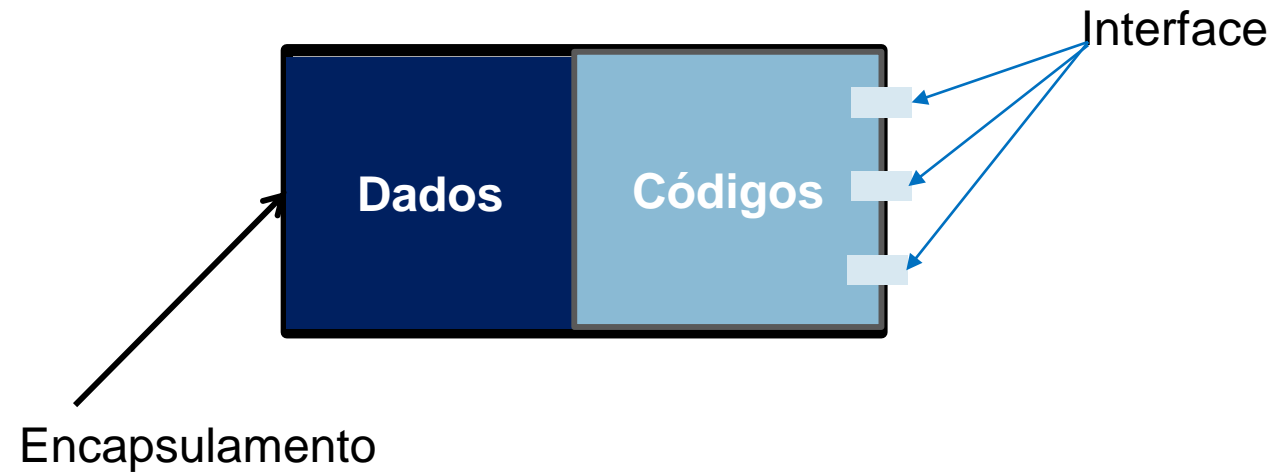
Prof. André Bernardi

andrebernardi@unifei.edu.br

Universidade Federal de Itajubá



CLASSES



Classes:

*“Estes tipos não são ‘abstratos’; eles são tão reais quanto um **int** ou um **float**.”*

(Doug McIlroy)

Alguns pontos importantes da aula passada:

1. Represente **conceitos** importantes para sua aplicação como classes, criando **novos tipos de dados**.
2. Separe a interface da classe (parte pública) de sua implementação (parte privada): encapsulamento.
3. Utilize dados públicos (*structs*) somente quando não houver restrições importantes com relação aos dados.
4. Defina sempre um construtor para a inicialização adequada de seus objetos.
5. Separe a implementação em arquivos de cabeçalho (**.h**) e código (**.cpp**).

Algumas vezes, uma função membro pode também ser definida no próprio arquivo de cabeçalho, mas neste caso, a função passa a ser **inline**, da mesma maneira que aprendemos com funções comuns. São chamadas *definições In-Class*.

Portanto, podemos dizer que este tipo de implementação somente deve ser utilizado para funções pequenas, raramente modificadas e que são utilizadas frequentemente.

Candidatos fortes seriam membros do tipo **get/set**, mais especificamente **get**, uma vez que membros **set** frequentemente precisam realizar validação em dados.

```

#ifndef DATE_H
#define DATE_H

class Date {
private:
    int d, m, y;
public:
    Date(int = 1, int = 1, int = 1972);
    ~Date() {} // In-Class

    // Definições In-Class de Funções
    // Repare que membros do tipo "get" são candidatos óbvios
    int get_day() { return d; }
    int get_month() { return m; }
    int get_year() { return y; }

    // No entanto, membros set somente são candidatos
    // quando não é necessário realizar validação nos dados.
    // Caso contrário, a definição no arquivo .cpp é mais apropriada.
    int set_day(int value) { d = value; }
    int set_month(int value) { m = value; }
    int set_year(int value) { y = value; }

};

#endif

```

Mutabilidade

Podemos declarar um objeto de uma classe tanto como uma **constante** (**const**) quanto como uma **variável**. Portanto, um identificador pode se referir a um objeto cujos valores sejam tanto “**imutáveis**”, quanto “**mutáveis**”.

A utilização sistemática de objetos imutáveis pode levar a códigos mais compreensíveis, facilitar a depuração do programa e aumentar até mesmo o desempenho.

Por este motivo, precisamos ser capazes de definir funções membro de classes que sejam aptas a trabalhar com objetos constantes: **funções membro constantes**.

```
// Exemplo de um objeto imutável  
const Date birth {7,11,1985};
```

Funções membro constantes são definidas utilizando-se a palavra **const** após a sua lista (vazia) de parâmetros, e **nunca podem modificar nenhum membro de dados da classe**. Veja exemplos:

```
// Todas as funções const se "comprometem"
// a não modificar o estado atual do objeto,
// provendo apenas acesso read-only.
int get_day() const { return d; }
int get_month() const { return m; }
int get_year() const { return y; }
void print() const;
```

Objetos declarados como variável podem acessar funções *const* e não *const*, enquanto **objetos imutáveis** somente podem acessar funções declaradas como **const**. Caso haja uma tentativa de acesso a membro não const, o compilador acusa erro:

```
principal1.cpp: In function 'int main()':
principal1.cpp:10:18: error: passing 'const Date' as 'this' argument of 'int Date::set_day(int)' discards qualifiers [-fpermissive]
    birth.set_day(10);
                   ^
```

Custo físico x custo lógico

Algumas vezes, uma função membro é logicamente **const**, mas ainda assim, é necessário que ela modifique algum membro de dados.

Mas como assim?

Algumas vezes, a chamada da função não resulta em uma modificação **visível** de estado para o usuário da classe, mas, internamente, alguma mudança é necessária. **Isso é custo lógico.**

Imagine, por exemplo, que haja uma necessidade de se contar quantas vezes um determinado objeto do tipo `Date` já tenha sido impresso. A contagem não muda os aspectos essenciais da data, mas teria que mudar o valor de um membro de dados **count**, p.ex.

Custo físico x custo lógico

Podemos definir um membro de dados que poderá ser modificado dentro de uma função *const* com a palavra reservada *mutable*. Veja:

```
mutable int count;
```

...

```
// imprime a data e conta a quantidade de chamadas
void Date::print() const
{
    count++;
    cout << d << "/" << m << "/" << y;
}
```

Tenha em mente que esta situação é relativamente incomum, e que apenas pequenas porções da classe devem ser declaradas como *mutable*.

Auto Referência

Nossas funções de atualização de valores (**set**) foram todas definidas como **void**, ou seja, sem retorno.

No entanto, pode ser interessante que este tipo de função retorne uma referência para o próprio objeto, de maneira que possamos **encadear** chamadas de funções. Podemos, por exemplo, querer escrever o seguinte:


```
// utilizando encadeamento de chamadas
Date today;
today.set_day(14).set_month(3).set_year(2016);
today.print();
```

Para isso ser possível, devemos modificar as implementações das funções membro **set**. Mas como acessar uma auto referência, ou seja, uma referência para o objeto que a chamou?

Use o ponteiro **this**!

```
Date& Date::set_day(int value)
```

```
{  
    // Esta primeira utilização do ponteiro de auto-referência  
    // é supérflua, ou seja, poderia ser descartada  
    this->d = value; // d = value; faria a mesma coisa (implicitamente)  
  
    // this é um ponteiro para o próprio objeto.  
    // Como devolvemos uma referência, precisamos retornar o valor (*)  
    return *this;  
}
```



```
Date& Date::set_month(int value)
```

```
{  
    this->m = value;  
    return *this;  
}
```

```
Date& Date::set_year(int value)
```

```
{  
    this->y = value;  
    return *this;  
}
```

Retornar uma referência para o próprio objeto chamador **não fere o encapsulamento**, pois mantém todas as relações de acesso.

Auto Referência

Um exemplo comum de utilização explícita do ponteiro **this** pode ser encontrado na manipulação de **listas encadeadas**.

Imagine uma classe Link, onde existe um método *insert*, que insere um valor antes dele.

```
class Link {  
    private:  
        Link* pre;  
        Link* suc;  
        int data;  
    public:  
        // construtor  
        Link(Link* p, Link* s, int d) : pre{p}, suc{s}, data{d} {}  
  
        // insere antes deste (de this)  
        Link* insert(int x)  
        {  
            return pre = new Link{pre, this, x}; // ...->pre->new->this->...  
        }  
};
```

Inicialização de membros de dados

Reparem em uma nova forma de inicialização dos membros de dados no construtor da classe *Link*, que é chamada de **lista inicializadora de membros**.

Qual é a sintaxe?

A lista inicia com dois pontos (:) e segue com os inicializadores individuais de cada membro separados por vírgula (,). Exemplos:

```
// construtor
Link(Link* p, Link* s, int d) : pre{p}, suc{s}, data{d} {}
```

```
// construtor
Date::Date(int dd, int mm, int yy) : d{dd}, m{mm}, y{yy}
{
    count = 0; // valor não recebido como parâmetro
}
```

Membros estáticos

Uma variável que faz parte da classe (escopo de classe), mas não faz parte dos objetos da classe (escopo de objeto), é uma variável estática, e a declaramos com a palavra reservada **static**.

Neste caso, **existe apenas uma cópia do membro estático em todo o programa**, ao invés de uma cópia para cada objeto, como acontece com membros de dados não-estáticos.

Este dado é **compartilhado por todas as instâncias daquela classe**, ou seja, por todos os objetos.

Mas qual a necessidade disso?

Imagine que se programa modela o Brasil hoje, e possui a classe Militante. Cada Militante tende a querer gritar uma frase de efeito, mas somente se houver pelo menos três militantes em seu grupo, para evitar ser hostilizado. Portanto, cada militante precisa conhecer a quantidade total de seus pares: **uma variável comum entre eles**. Uma informação que precisa ser **estática e compartilhada**.




```
#ifndef MILITANTE_H
#define MILITANTE_H

class Militante
{
    private:
        string nome;
        static int count;
    public:
        Militante(string);
        ~Militante() {}
        void grita();
};

#endif
```



```
int main()
{
    Militante m1 {"James"};
    m1.grita();

    Militante m2 {"Jake"};
    m2.grita();

    Militante m3 {"Jared"};
    m3.grita();
}
```

```
To quieto ainda: 1 militante(s)
To quieto ainda: 2 militante(s)
Eu tenho direito!
```

```
int Militante::count = 0; // é preciso inicializar aqui (Não pode ser In-Class)

Militante::Militante(string n) : nome{n}
{
    count++; // aumenta o numero de militantes
}

void Militante::grita()
{
    if(count >= 3)
        cout << "Eu tenho direito! \n";
    else
        cout << "To quieto ainda: " << count << " militante(s)\n";
}
```



Membros estáticos

Além de membros estáticos de dados, podem também haver funções membro estáticas. Este tipo de função poderá ser chamado mesmo quando não houver um objeto instanciado de sua classe. Ela não poderá acessar membros de dados não-estáticos da classe.

Veja um exemplo, utilizando a classe Circulo da aula passada:



```
// retorna o valor de area para um circulo
// qualquer cujo raio é passado como parametro
static float circle_area(float);
```



```
// Calcula a área de um círculo de raio r sem a
// necessidade de se criar um objeto da classe Circulo.
float Circle::circle_area(float r)
{
    return 3.141593 * r * r;
}
```



main()

```
// Acesso através do operador de escopo (::)
// Função estática é de escopo de classe, e não de instância
cout << "Area de um circulo de raio = 10: ";
cout << Circle::circle_area(10) << "\n";
```

Composição de Classes (tem um)

Durante a modelagem de seu sistema, você irá perceber que existe um número pequeno de classes que trabalham sozinhas. Na maior parte das vezes, o que acontece é uma **colaboração entre várias classes**, onde cada uma modela um aspecto diferente de seu universo.

Portanto, além de identificar os itens que formam o vocabulário do sistema, é necessário transformar cada um deles em um tipo (classe) e modelar os relacionamentos entre eles.

Outros tipos de relacionamentos serão vistos em disciplinas de “Engenharia de Software”. Por aqui, estaremos interessados em apenas dois: composição e generalização (que veremos em breve).

Exemplo: Suponha que precisamos lidar com dados pessoais, entre os quais **nome**, **endereço** e **data de nascimento**.

- Podemos ter uma classe para lidar com dados pessoais, por exemplo **Personal_data**.
- Como endereço e data de nascimento são elementos compostos, eles podem também ser representados como novos tipos de dados, ou classes **Date** e **Address**.
- Criamos membros de dados dos tipos Date e Address na classe Personal_data para representar a data de nascimento e o endereço da pessoa.

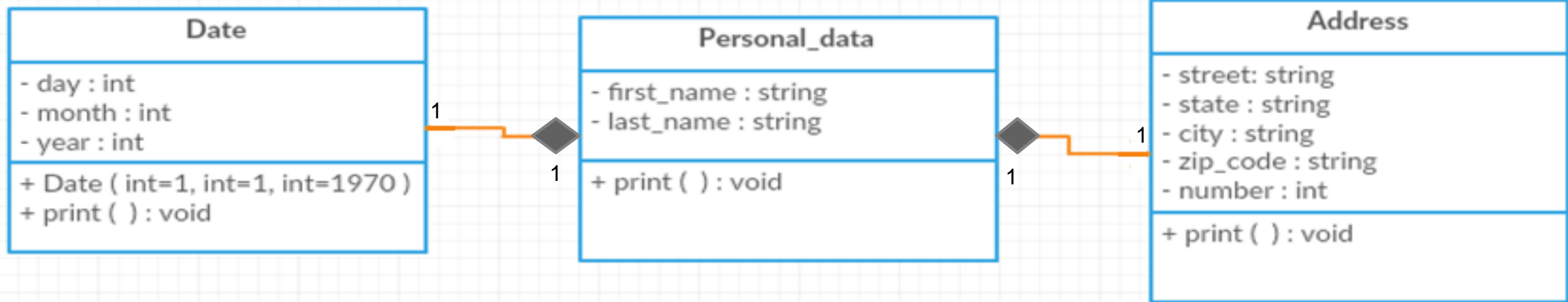
Este tipo de relacionamento é chamado de **composição**, e é **uma relação do tipo “tem um”**: a classe Personal_data será composta por membros das classes Date e Address, ou seja, terá um membro de cada tipo.

```
class Date {  
    private:  
        int day, month, year;  
    public:  
        Date(int=1, int=1, int=1970);  
        ~Date();  
        void print();  
        // ...  
};
```

```
class Address {  
    private:  
        string street, city, state, zip_code;  
        int number;  
    public:  
        Address();  
        ~Address();  
        void print();  
        // ...  
};
```

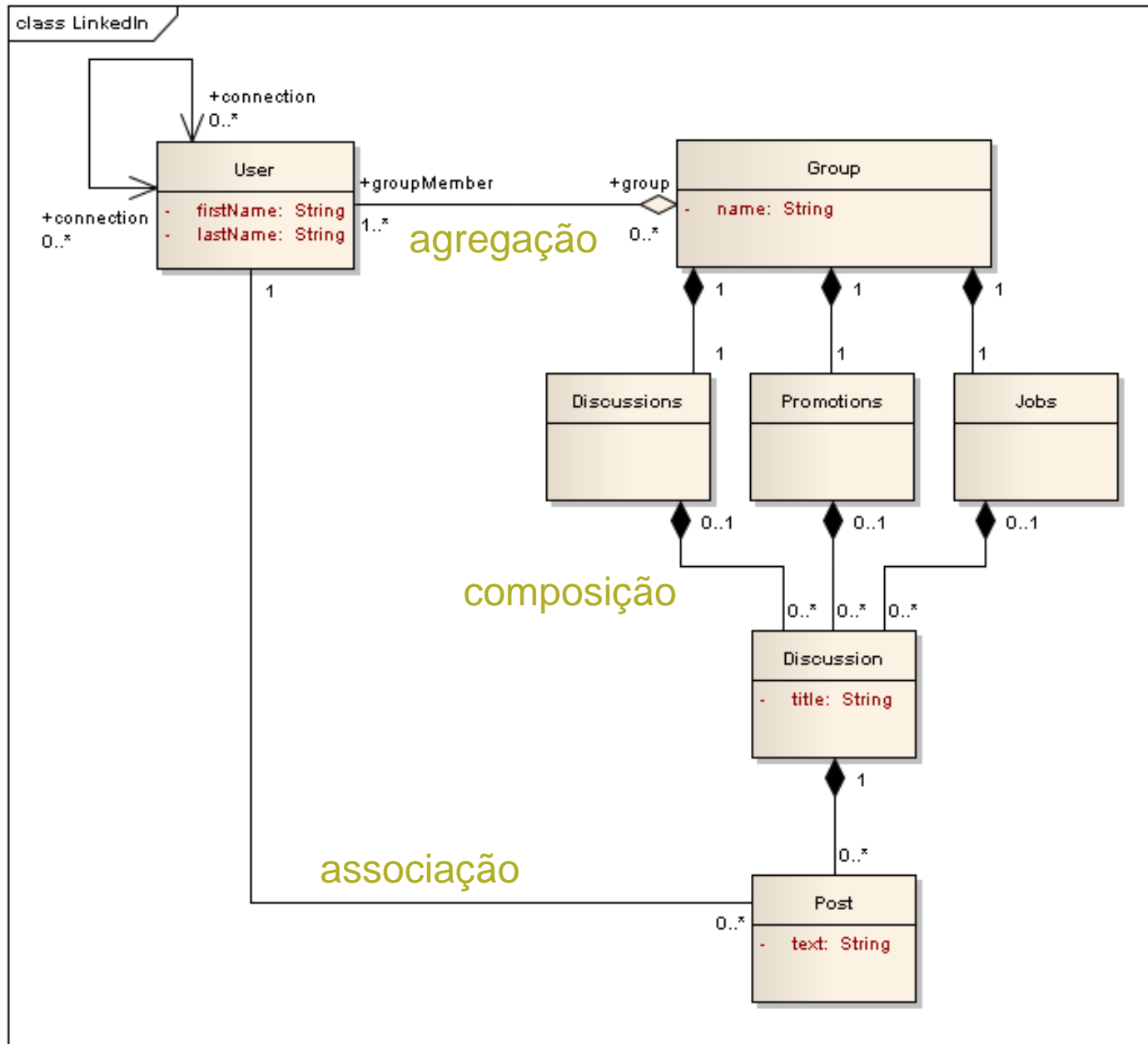
```
class Personal_data {  
    private:  
        string first_name, last_name;  
        int phone_number;  
        Date birth_date;  
        Address address;  
    public:  
        Personal_data();  
        ~Personal_data();  
        void print();  
        // ...  
};
```

Este tipo de interação entre classes é modelado de maneira diferente em diagramas de classes. Membros de dados cujo tipo tenha sido definido também pelo usuário entram como relacionamentos (conexões entre classes), e não são listadas no espaço dedicado aos membros de dados.



A **composição** é uma forma forte de agregação que requer que um objeto declarado como parte da classe seja incluído em, no máximo, um objeto composto por vez. Caso o objeto composto seja deletado, todas os seus objetos partes também são deletados com ele.

Existem vários detalhes sobre relacionamentos, que serão explorados propriamente nas disciplinas de Engenharia de Software. Veja um exemplo mais elaborado:



No diagrama de classes da rede social **LinkedIn** temos exemplos de alguns relacionamentos entre classes.

Temos alguns tipos de relacionamentos:

1. Associação: Uma classe possui um **ponteiro** para outra como membro de dados, sem gerenciar o ciclo de vida do mesmo. Portanto, a classe “sabe” sobre a outra.

Representada por uma **linha** que conecta classes.

2. Composição: Uma classe possui um **objeto** da outra como membro de dados. Portanto, a classe composta contém a outra, que não pode existir sem ela.

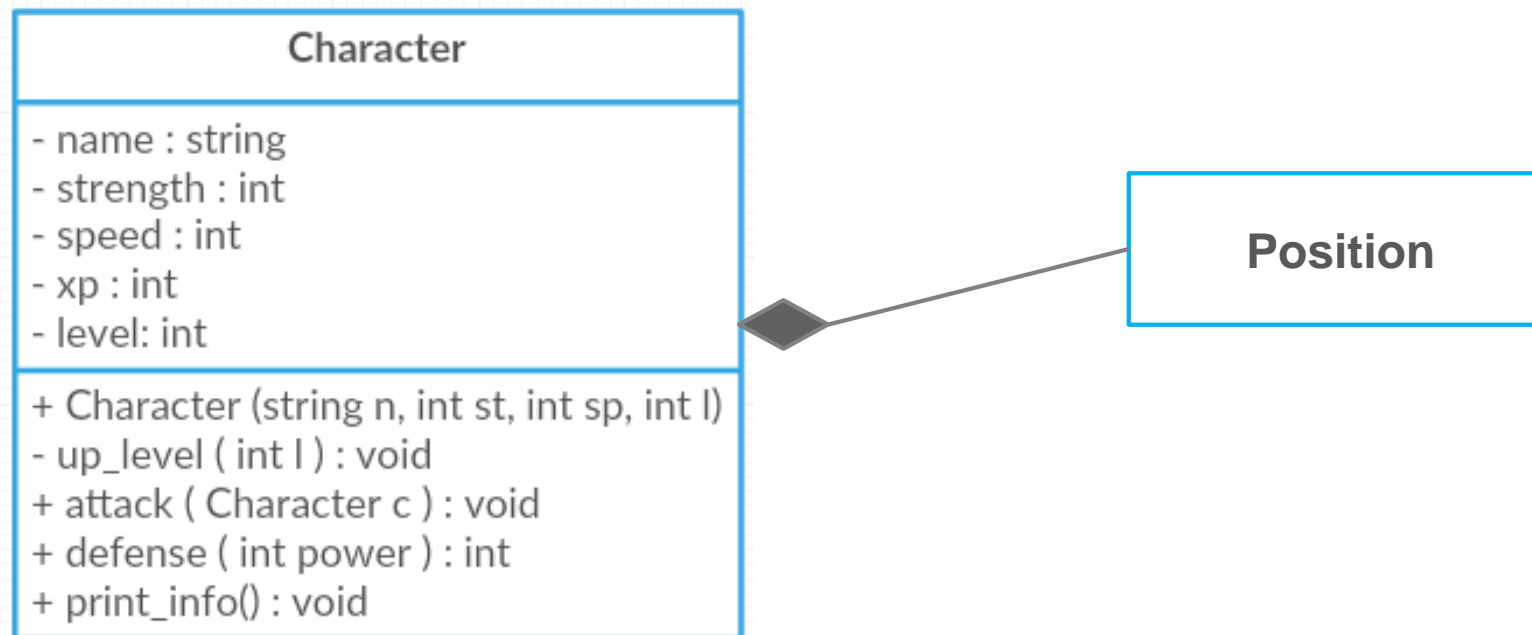
Representada por uma linha que conecta classes, com um **losango fechado** na extremidade da classe composta.

3. Agregação: Uma classe contém um **ponteiro** para outra, mas **gerencia** seu ciclo de vida. Ela contém a outra, mas esta também pode existir independentemente.

Representada por uma linha que conecta classes, com um **losango aberto** na extremidade da classe que contém a referência.

Exercício:

Implementar uma melhoria no sistema da aula passada, que contava com um personagem da classe **Character**. Agora, iremos acrescentar a ele uma posição em um determinado plano, que será modelada pela classe **Position**.



Referências

- <https://cplusplus.com/reference/>
- Notas de aula da disciplina Programação Orientada a Objetos, Prof. André Bernardi, Prof. João Paulo Reus Rodrigues Leite.