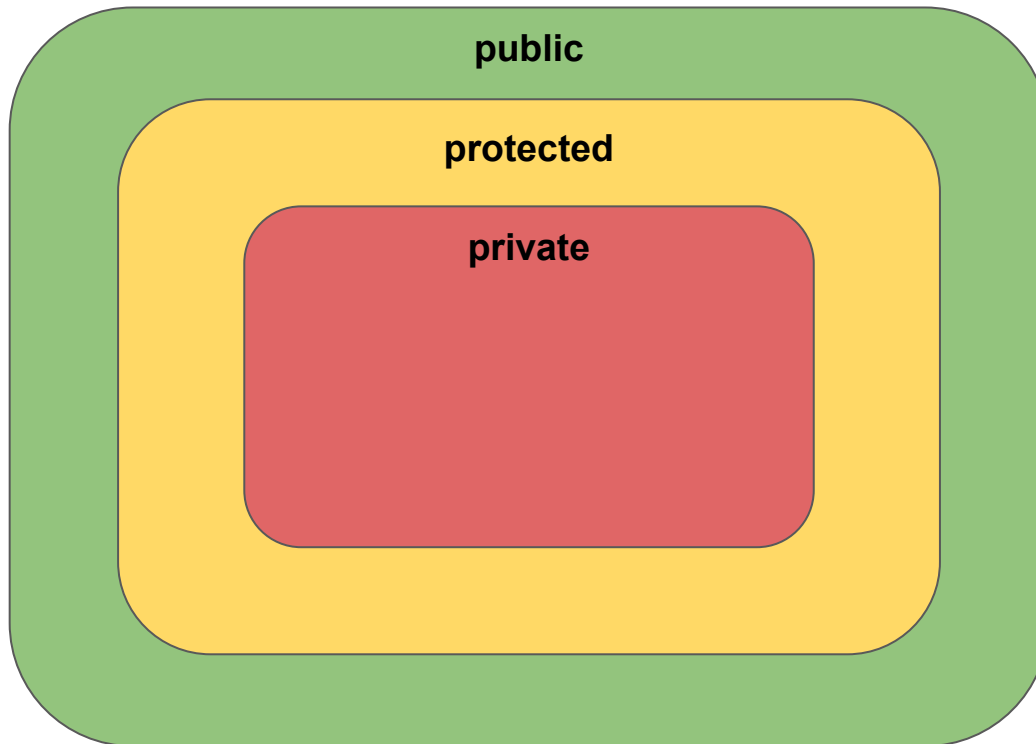


INF 112 - Programação II

Herança (Parte 2)

Encapsulamento

Modificadores de acesso



Protected

- Explora a Hierarquia de Classes
- Apenas subclasses podem acessar
- Não é visível para fora

Protected

```
class Base {  
protected:  
    int i = 0;  
};  
  
class Derived : public Base {  
public:  
    int f() {  
        i++;  
        return i;  
    }  
};
```

```
int main() {  
    Derived d1;  
    std::cout << d1.f() << std::endl;  
  
    Derived d2;  
    std::cout << d2.f() << std::endl;  
    return 0;  
}
```

Exemplo de protected



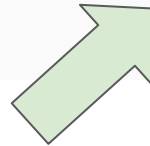
```
#include <string>

class Carta {
protected:
    int _dano;
public:
    Carta(int dano, std::string nome);
    virtual int get_dano();
    std::string get_nome();
};

int Carta::get_dano() {
    return this->_dano;
}
```

```
class CartaBoost : public Carta {
private:
    int _boost;
    std::string _nome;
public:
    CartaBoost(int boost, int dano,
               std::string nome);
    virtual int get_dano() override;
};

int CartaBoost::get_dano() {
    return Carta::_dano + _boost;
}
```



Exemplo de herança protected



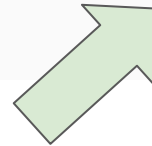
```
#include <string>

class Carta {
private:
    int _dano;
public:
    Carta(int dano, std::string nome);
    virtual int get_dano();
    std::string get_nome();
};

int Carta::get_dano() {
    return this->_dano;
}
```

```
class CartaBoost : public Carta {
private:
    int _boost;
    std::string _nome;
public:
    CartaBoost(int boost, int dano,
               std::string nome);
    virtual int get_dano() override;
};

int CartaBoost::get_dano() {
    return Carta::_dano + _boost;
}
```



Dano só é visível dentro de carta



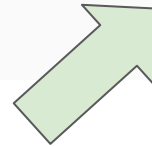
```
#include <string>

class Carta {
private:
    int _dano;
public:
    Carta(int dano, std::string nome);
    virtual int get_dano();
    std::string get_nome();
};

int Carta::get_dano() {
    return this->_dano;
}
```

```
class CartaBoost : public Carta {
private:
    int _boost;
    std::string _nome;
public:
    CartaBoost(int boost, int dano,
               std::string nome);
    virtual int get_dano() override;
};

int CartaBoost::get_dano() {
    return Carta::_dano + _boost;
}
```



Encapsulamento

Modificadores de herança

- Note que também definimos um modificador ao realizar a herança

```
class Base {  
    protected:  
        int i = 0;  
};  
  
class Derived : public Base {  
    public:  
        int f() {  
            i++;  
            return i;  
        }  
};
```


Encapsulamento

Modificadores de herança

- Indica **como** será herdado
- Diferente de **quais**. Sempre herdamos tudo que é **public, protected**

Encapsulamento

Modificadores de herança

- Indica **como** será herdado
- Diferente de **quais**
- Herdando como **public**
 - Caso comum
 - Herdamos todo o comportamento publicamente. Isto é, para o mundo externo somos do mesmo tipo

Encapsulamento

Modificadores de herança

- Indica **como** será herdado
- Diferente de **quais**
- Herdando como **private**
 - Raro. Prefira encapsulamento
 - Herdamos todo o comportamento em avisar para o mundo. Isto é, temos um estado similar, mas ninguém sabe qual é

Exemplo

Herança com Modificador

```
#include <string>
```

```
class Carta {  
private:  
    int _dano;  
public:  
    Carta(int dano, std::string nome);  
    virtual int get_dano();  
    std::string get_nome();  
};
```

```
class CartaBoost : public Carta {  
private:  
    int _boost;  
    std::string _nome;  
public:  
    CartaBoost(int boost,  
                std::string nome);  
    virtual int get_dano() override;  
};
```

```
int main() {  
    Carta c(2, "Monstro");  
    CartaBoost boost(3, "Boost");  
    boost.get_nome();  
}
```



Exemplo

Herança com Modificador

```
#include <string>
```

```
class Carta {  
private:  
    int _dano;  
public:  
    Carta(int dano, std::string nome);  
    virtual int get_dano();  
    std::string get_nome();  
};
```

```
class CartaBoost : private Carta {  
private:  
    int _boost;  
    std::string _nome;  
public:  
    CartaBoost(int boost,  
                std::string nome);  
    virtual int get_dano() override;  
};
```

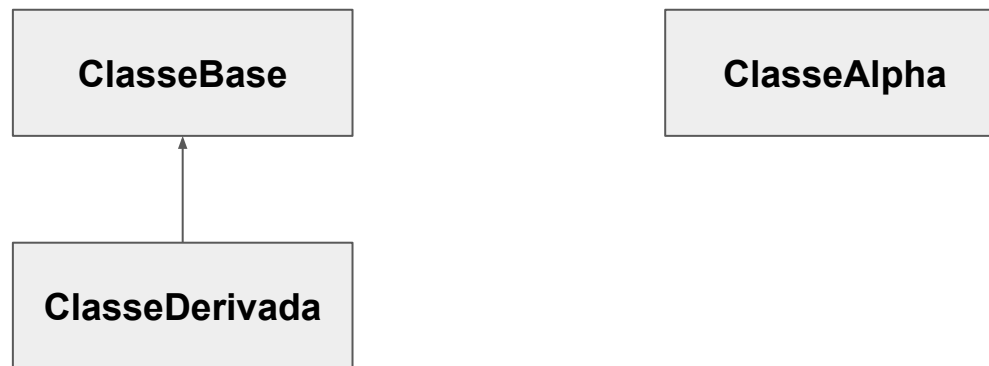
```
int main() {  
    Carta c(2, "Monstro");  
    CartaBoost boost(3, "Boost");  
    boost.get_nome();  
}
```



Encapsulamento

Modificadores de acesso

- Considerando a visibilidade dos membros da **ClasseBase** de acordo com o modificador de acesso



	ClasseBase	ClasseDerivada	ClasseAlpha
Public			
Protected			
Private			

Conversão de tipos

- Uma classe, ao herdar de outra, assume o tipo desta onde quer que seja necessário
- Upcasting
 - Conversão para uma classe mais genérica
- Downcasting
 - Conversão para uma classe mais específica

Conversão de tipos

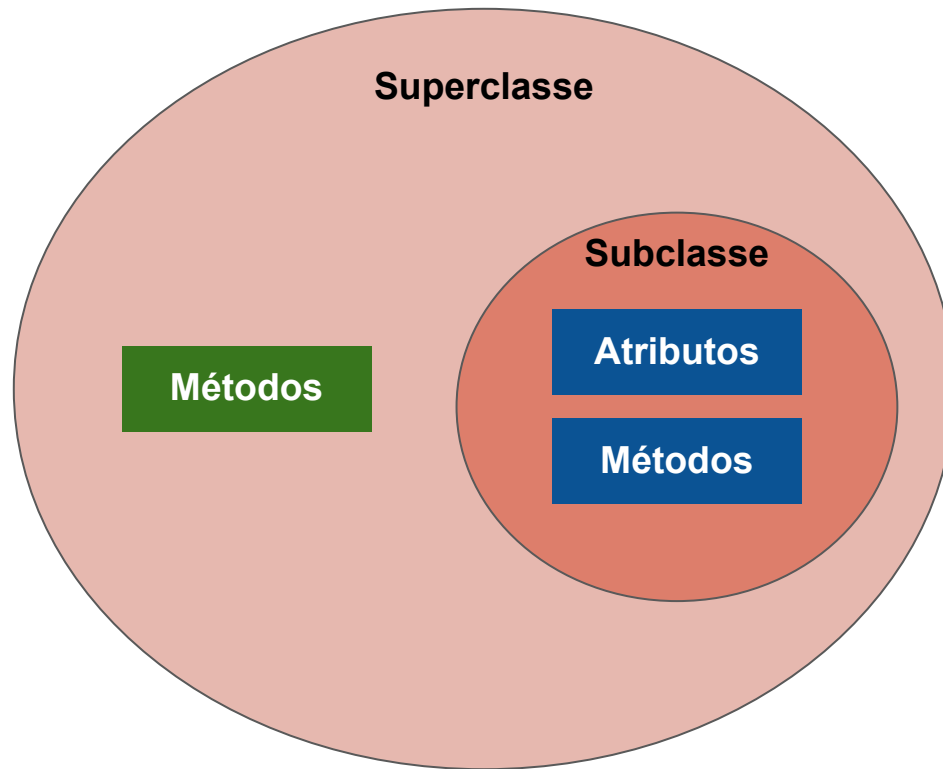
Upcasting

- Ocorre no sentido Classe \Rightarrow Superclasse
- Não há necessidade de indicação explícita
- A classe derivada sempre vai manter as características públicas da superclasse

Conversão de tipos

Upcasting

Contexto de Classe



Upcasting

Note que os objetos viram Pessoa auto-magicamente

```
#include <iostream>

#include "estudante.h"
#include "pessoa.h"

void f(Pessoa &pessoa) {
    std::cout << "Na função: " << pessoa.defina_meu_tipo() << std::endl;
}

int main() {
    Pessoa pessoa("Julio Reis.");
    Estudante estudante("Jane Doe", 20180101);
    std::cout << "A pessoa é: " << pessoa.defina_meu_tipo() << std::endl;
    std::cout << "O estudante é: " << estudante.defina_meu_tipo() << std::endl;
    f(pessoa);
    f(estudante);
    return 0;
}
```



Conversão de tipos

Downcasting

- Ocorre no sentido Subclasse \Rightarrow Classe
- Não é feito de forma automática!
- Deve-se deixar explícito, informando o nome do subtipo antes do nome da variável

Conversão de tipos

Downcasting

- Nem sempre uma superclasse poderá assumir o tipo de uma subclasse
- Todo Aluno é uma Pessoa
- Nem toda Pessoa é Professor
- Caso não seja possível
 - C++ é esquisito, o erro ocorre no futuro.
 - Use `<dynamic_cast>`

Dynamic Cast

A primeira conversão é ok

```
#include <iostream>


#include "estudante.h"
#include "pessoa.h"

int main() {
    Pessoa pessoa("Julio Reis.");
    Estudante estudante("Julio Reis Doe", 20180101);

    Pessoa &p2 = estudante;
    std::cout << p2.defina_meu_tipo() << std::endl;

    Estudante &e2 = dynamic_cast<Estudante&>(p2);

    return 0;
}
```



Dynamic Cast

A segunda é explícita

```
#include <iostream>


#include "estudante.h"
#include "pessoa.h"

int main() {
    Pessoa pessoa("Julio Reis.");
    Estudante estudante("Jane Doe", 20180101);

    Pessoa &p2 = estudante;
    std::cout << p2.defina_meu_tipo() << std::endl;

    Estudante &e2 = dynamic_cast<Estudante&>(p2);

    return 0;
}
```



Dynamic Cast

- Mais Seguro em C++
- Você pode fazer um cast <tipo> X
 - Sem usar `dynamic_cast`
 - Chamado de um static cast
- Para entender a necessidade é bom entender como funciona polimorfismo

Exemplo Abaixo


Note que removemos a referência. Qual a saída?!

```
#include <iostream>

#include "estudante.h"
#include "pessoa.h"

int main() {
    Pessoa pessoa("Julio Reis.");
    Estudante estudante("Julio Reis. Doe", 20180101);

    Pessoa p2 = estudante;
    std::cout << p2.defina_meu_tipo() << std::endl;
    return 0;
}
```



Polimorfismo e Late Binding

- Polimorfismo funciona bem quando fazemos uso de referências e ponteiros
- Um ponteiro nada mais é do que um endereço um inteiro, então com `dynamic_cast` a linguagem verifica se é possível
- Sem isso, C++ faz a conversão e deixa acontecer

Exemplo Abaixo

Com ponteiros funciona

```
#include <iostream>

#include "estudante.h"
#include "pessoa.h"

int main() {
    Pessoa pessoa("Julio Reis.");
    Estudante estudante("Julio Reis. Doe", 20180101);

    Pessoa *p2 = &estudante;
    std::cout << p2->defina_meu_tipo() << std::endl;
    return 0;
}
```



Dynamic Cast

Caso dê errado, e2 é nullptr

```
#include <iostream>

#include "estudante.h"
#include "pessoa.h"

int main() {
    Pessoa pessoa("Julio Reis.");
    Estudante estudante("Jane Doe", 20180101);

    Pessoa *p2 = &estudante;
    std::cout << p2->defina_meu_nome() << std::endl;

    Estudante *e2 = dynamic_cast<Estudante*>(p2);

    return 0;
}
```



Static Cast

Caso dê errado, e2 é lixo

```
#include <iostream>

#include "estudante.h"
#include "pessoa.h"

int main() {
    Pessoa pessoa("Julio Reis.");
    Estudante estudante("Jane Doe", 20180101);

    Pessoa *p2 = &estudante;
    std::cout << p2->defina_meu_nome() << std::endl;

    Estudante *e2 = <Estudante*> p2;

    return 0;
}
```



Nas próximas aulas

- Interfaces
- Classes Abstratas
- + Polimorfismo