

A Review of the OOP Polymorphism Concept



Outline

- Overview
- Type Casting and Function Ambiguity
- Virtual Functions, Pure Virtual Functions, and Abstract Class
- Non Type Casting Programming, and Scalability Problems
- Class Composition Framework
- Summary

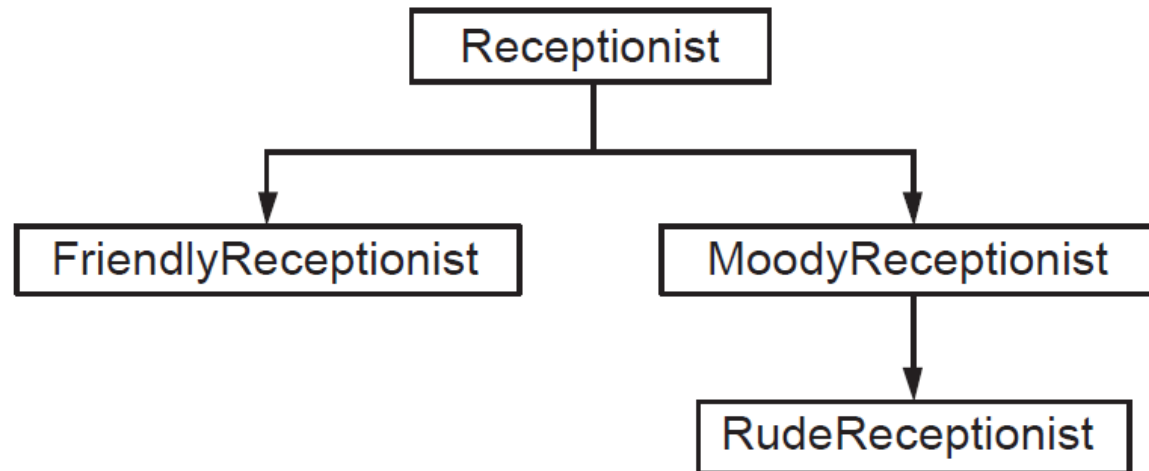
Polymorphism: Overview

- An important concept in object oriented programming (OOP)
- A polymorphic function
 - acts differently under different context.
 - has different implementation under different context.



Polymorphism: Inheritance

- Receptionist and how they greet customers



- Friendly: "Good morning. How can I help you today?"
- Moody: "What do you want?"
- Rude: "What do you want? I'm busy. Come back later"

Polymorphism: Ex 1

```
class Receptionist {
    public:
    void greet() {cout<<"Say:\n";};
};

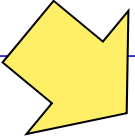
class FriendlyReceptionist : public Receptionist {
    public:
    void greet(){
        cout<<"Say: Good morning. How can I help you today?\n"
    }
};

class MoodyReceptionist : public Receptionist {
    public:
    void greet() { cout<<"Say: What do you want?\n"; };
};

class RudeReceptionist : public MoodyReceptionist {
    public:
    void greet(){
        MoodyReceptionist::greet();
        cout<<"Say: I'm busy. Come back later.\n";
    }
};
```

Polymorphism: Example

```
main() {
    FriendlyReceptionist f_obj;
    MoodyReceptionist m_obj;
    RudeReceptionist r_obj;
    cout<<"\n----- Friendly Receptionist ---\n";
    f_obj.greet();
    cout<<"\n----- Moody Receptionist -----\n";
    m_obj.greet();
    cout<<"\n----- Rude Receptionist -----\n";
    r_obj.greet();
    cout<<"-----\n";
}
```



```
>>./receptionist
----- Friendly Receptionist -----
Say: Good morning. How can I help you today?

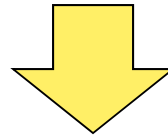
----- Moody Receptionist -----
Say: What do you want?

----- Rude Receptionist -----
Say: What do you want?
Say: I'm busy. Come back later!!
-----
```

Polymorphism: Ex 2

- **Modify class** MoodyReceptionist

```
class MoodyReceptionist : public Receptionist {  
    public:  
void greet() { cout<<"Say: What do you want?\n"; };  
};
```

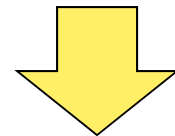


```
>>./receptionist  
----- Friendly Receptionist -----  
Say: Good morning. How can I help you today?
```

```
----- Moody Receptionist -----  
Say:
```

```
----- Rude Receptionist -----  
Say:  
Say: I'm busy. Come back later!!  
-----
```

Why?:



INHERITANCE

Outline

- Overview
- Type Casting and Function Ambiguity
- Virtual Functions, Pure Virtual Functions, and Abstract Class
- Non Type Casting Programming, and Scalability Problems
- Class Composition Framework
- Summary

Polymorphism: Ex 3

- Ex1:

```
main() {
    FriendlyReceptionist f_obj;
    MoodyReceptionist m_obj;
    RudeReceptionist r_obj;
    cout<<"\n----- Friendly Receptionist ---\n";
    f_obj.greet();
    cout<<"\n----- Moody Receptionist -----\n";
    m_obj.greet();
    cout<<"\n----- Rude Receptionist -----\n";
    r_obj.greet();
    cout<<"-----\n";
}
```

- Three variables, Three classes
- No type casting!

Polymorphism: Ex 3

- Based on Ex1: Let implement type casting

```
main() {  
    FriendlyReceptionist *f_pt;  
    MoodyReceptionist *m_pt, *r_pt;  
    f_pt = new FriendlyReceptionist();  
    m_pt = new MoodyReceptionist();  
    r_pt = new RudeReceptionist();  
  
    cout<<"\n----- Friendly Receptionist ----\n";  
    f_pt->greet();  
    cout<<"\n----- Moody Receptionist ----\n";  
    m_pt->greet();  
    cout<<"\n----- Rude Receptionist ----\n";  
    r_pt->greet();  
    cout<<"-----\n";  
}
```



Polymorphism: Ex 3

- After running:

```
>>./receptionist
```

```
----- Friendly Receptionist -----
```

```
Say: Good morning. How can I help you today?
```

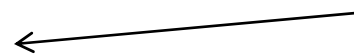
```
----- Moody Receptionist -----
```

```
Say: What do you want?
```

```
----- Rude Receptionist -----
```

```
Say:
```

Q: ?



```
class MoodyReceptionist : public Receptionist {
public:
void greet() { cout<<"Say: What do you want?\n"; };
};
```

```
class RudeReceptionist : public MoodyReceptionist {
public:
void greet(){
MoodyReceptionist::greet();
cout<<"Say: I'm busy. Come back later.\n";
};
};
```

Why?

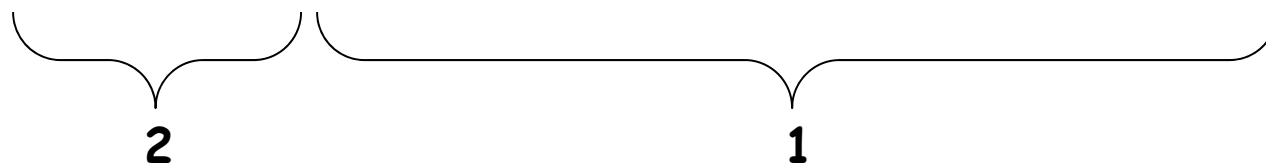
→ type casting

Polymorphism: Ex 3

- `*r_pt` is of class `MoodyReceptionist`
`MoodyReceptionist *m_pt, *r_pt;`

- Two steps process when invoking

```
r_pt = new RudeReceptionist();
```



1. Create a `RudeReceptionist` object
2. Cast the created object to be a `MoodyReceptionist` object

Polymorphism: Ex 3

- The object `*r_pt` was doing what Moody Receptionist does
- What if we want the object `*r_pt` what RudeReceptionist do ?
 1. Declare `*r_pt` as RudeReceptionist
→ Scalability problem!!
 2. USE VIRTUAL FUNCTION



Outline

- Overview
- Type Casting and Function Ambiguity
- Virtual Functions, Pure Virtual Functions, and Abstract Class
- Non Type Casting Programming, and Scalability Problems
- Class Composition Framework
- Summary

Virtual Functions

- Carry inheritance through type casting.
- Bind the implementation
 - to the "construction type",
 - **not** the "declaration type"
- E.g.,

```
MoodyReceptionist *r_pt;  
r_pt = new RudeReceptionist();
```

Which implementation of
a virtual function `greet()` will be used for
`r_pt->greet()`?

RudeReceptionist

Virtual Functions

- Why this is a good idea?
- Declare a very general pointer:
 - `Receptionist *a,*b;`
- Create the object on the fly as you wish
 - `a = new RudeReceptionist();`
 - `b = new FriendlyReceptionist();`
- At **declaration**, you do not need to think what type of receptionist you want.
- Decide later at the **construction**.



Polymorphism: Ex 3

- After running:

```
>>./receptionist
----- Friendly Receptionist -----
Say: Good morning. How can I help you today?

----- Moody Receptionist -----
Say: What do you want?

----- Rude Receptionist -----
Say: What do you want?
```

```
class MoodyReceptionist : public Receptionist {
public:
void greet() { cout<<"Say: What do you want?\n"; };
};
```

```
class RudeReceptionist : public MoodyReceptionist {
public:
void greet(){
MoodyReceptionist::greet();
cout<<"Say: I'm busy. Come back later.\n";
};
};
```

Virtual Functions: Ex4

- Ex 3:

```
>>./receptionist
----- Friendly Receptionist -----
Say: Good morning. How can I help you today?

----- Moody Receptionist -----
Say: What do you want?

----- Rude Receptionist -----
Say: What do you want?
```

```
class MoodyReceptionist : public Receptionist {
public:
    void greet() { cout<<"Say: What do you want?\n"; };
};
```



Virtual Functions: Ex4

- Based on Ex 3, modify class `Receptionist`

```
class Receptionist {  
    public:  
    virtual void greet() {cout<<"Say:\n";}  
};
```

- At run time,

```
>>./receptionist  
----- Friendly Receptionist -----  
Say: Good morning. How can I help you today?  
  
----- Moody Receptionist -----  
Say: What do you want?  
  
----- Rude Receptionist -----  
Say: What do you want?  
Say: I'm busy. Come back later!!  
-----
```

Note:

- Virtuality is inheritable.
- You only need to do it once at the base class.

Pure Virtual Functions

- Make virtuality mandatory
- Ex5: From Ex4, modify class Receptionist

```
class Receptionist {  
    public:  
    virtual void greet()=0;  
};
```

- Obtain the same results.

```
>>./receptionist  
----- Friendly Receptionist -----  
Say: Good morning. How can I help you today?  
  
----- Moody Receptionist -----  
Say: What do you want?  
  
----- Rude Receptionist -----  
Say: What do you want?  
Say: I'm busy. Come back later!!  
-----
```

Pure Virtual Functions

- What's difference?
- Implementation is now mandatory.
- If no implementation in the derived class,
 - Virtual: Use that of the base class
 - Pure virtual: The class is non-instantiable!!; An attempt → **compilation error**
- Summary

		Base Class	Derive Class
Virtual	Implementation	Provide	Optional
	Declaration	Virtual	Optional
Pure virtual	Implementation	None	Mandatory
	Declaration	Virtual, =0	Mandatory

Abstract Class

- A class with at least one pure virtual function.
- **Incomplete; Non-instantiable.**
- Only have "what to do"
- Derive class providing "how to do" is **complete** and **instantiable**.
- The use of polymorphism has 3 main components:
 1. A pure virtual function
 2. An abstract class
 3. An instantiable class



Pure Virtual Functions

- Last catch: Related declaration

Declaration	Example
Pure virtual declaration	<code>virtual void greet()=0;</code>
Declaration with no action	<code>virtual void greet() {};</code>
Invalid declaration	<code>virtual void greet();</code>

- If the base class is abstract, you cannot leave the derived class unimplemented.
- Otherwise → compilation error for object instantiation.



Outline

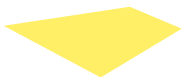
- Overview
- Type Casting and Function Ambiguity
- Virtual Functions, Pure Virtual Functions, and Abstract Class
- Non Type Casting Programming, and Scalability Problems
- Class Composition Framework
- Summary

Type Casting v.s. Non-Type Casting

- The ambiguity occurs due to type casting:

```
MoodyReceptionist *r_pt;  
r_pt = new RudeReceptionist();
```

- Can we not use type casting? → Yes
- Then, what's problem?
- Type casting programming is more elegant and scalable!!



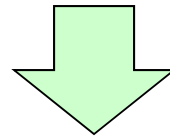
Non-Type Casting Programming

- Let make the receptionist concept more interesting:
 - Create a company,
 - Employ a receptionist, and
 - Serve customers
- About the company:
 - Base class `Company`:
 - Derived class `MoodyCompany`: Employ a moody receptionist.



Non-Type Casting Programming: Ex6

```
class Company {  
    public:  
        void serve() {  
            greet();  
            cout<<"\nServing the customer ... \n";  
        };  
        void greet () {};  
};
```



```
class MoodyCompany : public Company {  
    public:  
        MoodyCompany(){employee_ = new MoodyReceptionist;};  
        void greet(){employee_->greet();};  
    private:  
        MoodyReceptionist* employee_  
};
```

Non-Type Casting Programming: Ex6

```
int main() {  
    MoodyCompany my_company;  
    my_company.serve();  
    return 0;  
}
```

`greet();`

Q: Why saying this?

```
>>./company  
Say: What do you want?  
Serving the customer ...
```

`cout<<"Serving the customer..."`

- Here, we do not use virtuality.
- Do we need one? What if we do not use it?

A Scalability Problem

Q: What do we do if we want have the company to greet customer nicely?

A: Put a friendly receptionist in the company

Q: How do we do that in C++?

A:

```
class FriendlyCompany : public Company {
public:
    FriendlyCompany(){employee_ = new FriendlyReceptionist;};
    void greet(){employee_->greet();};
private:
    FriendlyReceptionist* employee_;
};
```

A Scalability Problem

Q: What's problem?

A: We do not reuse the existing codes!

Q: How many company class do we need (excluding the base class) for 3 types of receptionist?

A: ()

Q: What if we have 3 types of receptionist, 5 types of engineers, and 10 types of accountants. How many classes?

A: ()

Outline

- Overview
- Type Casting and Function Ambiguity
- Virtual Functions, Pure Virtual Functions, and Abstract Class
- Non Type Casting Programming, and Scalability Problems
- Class Composition Framework
- Summary

Class Composition Framework

- A solution to the scalability problem
- Consider the previous example: Companies with 3 receptionists, 5 engineers, 10 accountants
- Class composition defines
 - One company
 - `Class Company`
 - Put people in company without deriving the company class
 - Use function `hire(...)`



Class Composition Framework: Ex7

```
class Company {
public:
    void serve() {
        recp_->greet();
        cout<<"\nServing
            the customer ... \n";
    };
    void hire(Receptionist* r) {
        recp_ = r;
    };
    void hire(Engineer* e) {
        engr_ = e;
    };
private:
    Receptionist* recp_;
    Engineer* engr_;
};
```

```
int main() {
    MoodyReceptionist *m_pt= new
        MoodyReceptionist();
    GoodEngineer *e_pt = new
        GoodEngineer();
    Company my_company;
    my_company.hire(m_pt);
    my_company.hire(e_pt);
    my_company.serve();
    return 0;
}
```

Class Composition Framework

- Come back to our problem.
- We have company with 3 receptionist, 5 engineers, 10 accountants

Q: How many company classes do we need (excluding employee's classes)?

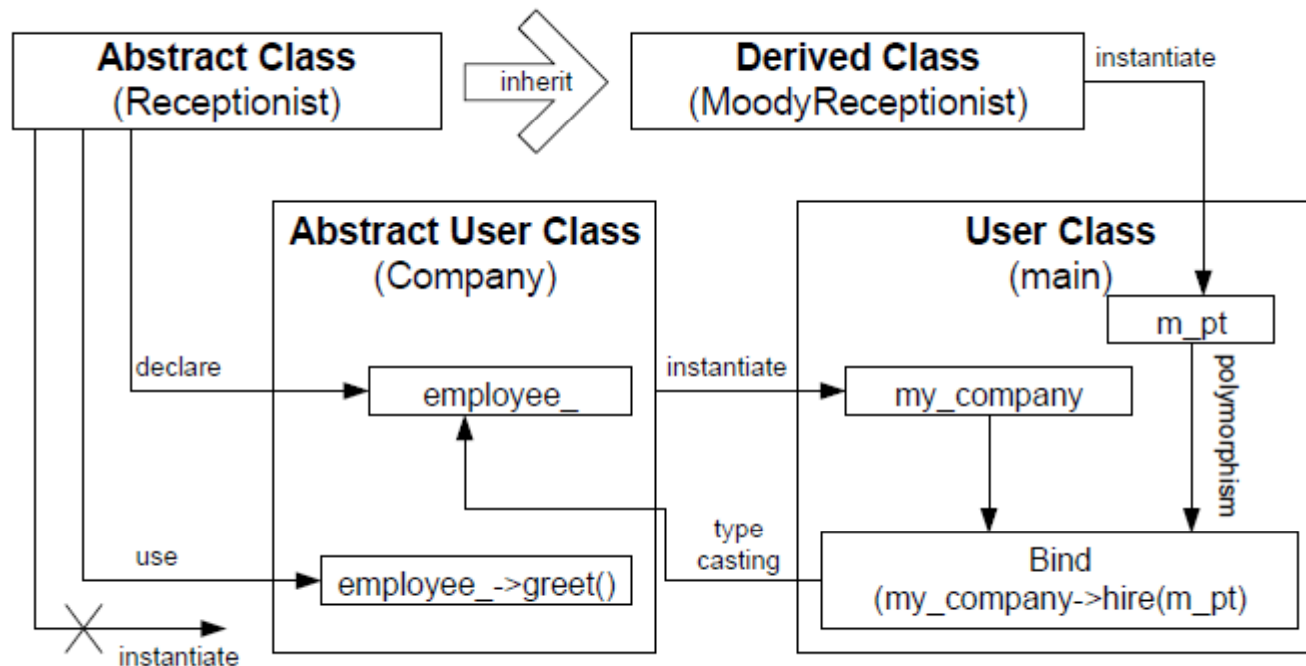
A: No Type Casting: ()

A: Class composition: ()



Class Composition Framework: Main Components

1. An abstract class: Receptionist
2. An derived class: MoodyReceptionist
3. An abstract user class: Company
4. A user class: main()



Class Composition Framework

```
class Company {
public:
    void serve() {
        recp_->greet();
        cout<<"\nServing
            the customer ... \n";
    };
    void hire(Receptionist* r) {
        recp_ = r;
    };
    void hire(Engineer* e) {
        engr_ = e;
    };
private:
    Receptionist* recp_;
    Engineer* engr_;
};
```

**From Ex7,
What do we have to do to
have the company greet
customers nicely?**

```
int main() {
    MoodyReceptionist *m_pt= new
        MoodyReceptionist();
    GoodEngineer *e_pt = new
        GoodEngineer();

    Company my_company;
    my_company.hire(m_pt);
    my_company.hire(e_pt);
    my_company.serve();
    return 0;
}
```

Outline

- Overview
- Type Casting and Function Ambiguity
- Virtual Functions, Pure Virtual Functions, and Abstract Class
- Non Type Casting Programming, and Scalability Problems
- Class Composition Framework
- Summary

Summary

- Polymorphism:
 - An important OOP concept
 - Act differently under different context
- Example--Receptionist: Friendly, Moody, Rude
- Type casting problem
 - Regular function \rightarrow () type
 - Virtual function \rightarrow () type



Summary

- Pure virtual function:

()

- Abstract class

- Definition: ()

- Usage: ()



Summary

- Scalability of non-type casting programming
- Class composition framework
 - Abstract class
 - Derived class
 - Abstract user class
 - User class

