

POLYMORPHISM

There are three major concepts in object-oriented programming:

1. Encapsulation (Classes, Objects)

Data and related functions are placed into the same entity.

Provides **data abstraction** and **information hiding** (public: interface, private: implementation)

2. Inheritance

Is-a relation, generalization-specialization

Promotes code reusability.

Enables the use of common interfaces for different classes (types).

3. Polymorphism (dynamic polymorphism, subtyping, or inclusion)

The derived class can override the methods of the base class.

Allows the object type to determine which specific implementation of a method to call at run time (dynamic method binding).

Requires inheritance between classes.

Improves the design with common interfaces.

In these lecture slides, what we refer to as "polymorphism" is runtime, dynamic polymorphism, formally known as subtyping (or inclusion polymorphism).

Types of Polymorphism:

Polymorphism means "taking many shapes".

- Programming language theory defines various forms of polymorphism.
- Definition given by Bjarne Stroustrup:
"Polymorphism is providing a **single interface** to entities of **different types**.
Virtual functions provide dynamic (run-time) polymorphism through an interface provided by a base class.
Overloaded functions and templates provide static (compile-time) polymorphism."
- In general, polymorphism is calling different functions with the same name based on the type of the related objects (the object for which the function is called and the parameters).

Two types of polymorphism:

A) Static (compile-time) polymorphism:

- Ad hoc polymorphism: function and operator overloading
Examples: `void print(int); void print (std::string);` or `obj1 + obj2; obj1 + 5;`
- Parametric polymorphism: function and class templates
Generic programming (see Chapter 09)

B) Dynamic (run-time) polymorphism:

- Which method to call is determined at **runtime** based on the type of the object that receives the message.

Polymorphism in real life:

- In real life, there is often a collection of different objects that, given identical instructions (messages), should take different actions based on their types.

Example: The dean is a professor. Professors and deans visit the rector.

- Sometimes, professors and deans may visit the university's rector.
- The rector is also a professor, but we will ignore this relationship for this example.
- When the rector meets with a visitor, they ask the visitor to print their information.
- The rector sends the same `print()` message to a professor or dean.
- Different types of objects (professor or dean) have to print different information.
- The rector **does not know the type** of visitor (professor or dean) and always sends the **same** `print()` message.
- Depending on the **type** of visitor (receiving object), **different actions** are performed.
- The same message (`print`) works for everyone because everyone knows how to print their information.
- The rector's single instruction is polymorphic because it works differently for different kinds of academic staff.

Dynamic (runtime) polymorphism in programming:

- In C++, dynamic (runtime) polymorphism means that a call to a member function will cause a **different function** to be executed depending on the **type of object** that receives the message.
- In runtime polymorphism, **the sender of the message does not need to know the type of the receiving object at compile time**.
- Dynamic polymorphism occurs in classes that are related by **inheritance**.

Example:

- Remember: A pointer (or reference) to base (e.g., Professor) can also point to derived (e.g., Dean) objects because Dean is a Professor.

```
Professor *ptr;           // ptr can point to Professor and Dean objects
...                       // The address pointed to by ptr will be determined at runtime
if (condition) ptr = &professor_obj;
else ptr = &dean_obj;

ptr->print();             // which print method at compile time (professor or dean)?
```

- If `print()` is a polymorphic function, the decision of which function to call will be made at **runtime** based on the type of object pointed to by the pointer `ptr`.

Calling redefined, nonvirtual member functions using pointers (name hiding, no polymorphism)

- The base and derived classes have methods with the same signature (name and parameters).
- The methods are **not virtual (no dynamic polymorphism)**.
- We access methods of the base and derived classes using pointers.

Example: Professors and deans visit the rector

```
class Professor{                                // Base class: Professor
public:
    void print() const;
    :
};
```

```
class Dean : public Professor{    // Derived class: Dean
public:
    void print() const;          // redefined
    :
};
```

- Both classes have a function with the same signature: `print()`.
- They print different information.
 - Professor: name and research area.
 - Dean: name, research area, and faculty name.
- In this example, these functions are not virtual (**not polymorphic**).

Calling redefined, nonvirtual member functions using pointers (cont'd)

Example (cont'd): Professors and deans visit the rector

```
class Rector {                // User class: Rector
public:
    void meetVisitor(const Professor*) const;
};
```

Association between Rector and Professor

A pointer to the base class

```
// The input parameter is a pointer to Professor (Base)
void Rector::meetVisitor(const Professor* visitor) const
{
    visitor->print();           // which print?
}
```

Since the input parameter is a pointer to the Professor (base) class, we can call this method by sending the address of a Professor object or the address of a Dean object.

The visitor can be any professor, e.g., department head or dean.

Calling Redefined, nonvirtual member functions using pointers (cont'd)**Example (cont'd):** Professors and deans visit the rector

```

int main(){
    Rector itu_rector;
    Professor prof1("Professor 1", "Robotics");
    Dean dean1("Dean 1", "Computer Networks", "Engineering Faculty");

    Professor *ptr;                // A pointer to Base type
    char input_char;
    std::print("Professor or Dean (p/else)); std::cin >> input_char;
    if (input_char == 'p') ptr = &prof1;    // ptr points to a professor
        else ptr = &dean1;    // ptr points to a dean
    itu_rector.meetVisitor(ptr);    // which print?

```

Example e08_1a.cpp

- In this example, when the statement visitor->print() calls print(), the print() function of the base class (Professor) gets executed in both cases.
- Professor::print() is invoked for both of the objects (prof1 and dean1).
- The compiler ignores the **contents** of the pointer and chooses the (nonvirtual) member function that matches the **type** of the pointer. meetVisitor(const Professor* visitor)

Since the methods are **not virtual**, the decision in meetVisitor is made at **compile time**.
The same print() function is invoked for all types. This is not polymorphism!

Calling redefined, virtual member functions using pointers (Dynamic Polymorphism)

- We make a single change in the program e08_1a.cpp and place the keyword virtual in front of the declaration of the print() function in the base class.

Example:

```

class Professor{                // Base class: Professor
public:
    virtual void print() const;    // A virtual (polymorphic) function
};

class Dean : public Professor{   // Derived class: Dean
public:
    void print() const;          // It is also virtual (polymorphic)
};

```

The virtual keyword is optional (not mandatory) for the derived class.
 If a method of Base is virtual, the redefined method in Derived is also virtual.

```

// The input parameter is a pointer to Professor (Base) class
void Rector::meetVisitor(const Professor* visitor) const
{
    visitor->print();            // We did not change the methods of Rector
                                // Which print?
}

```

Example e08_1b.cpp

Calling redefined, virtual member functions using pointers (Polymorphism) (cont'd)

- Since the print() functions are virtual, **functions** are executed depending on the **contents** of the pointer, **rather than its type**.
- The decision is made at **runtime** for visitor->print().

Virtual (polymorphic) functions are called based on the **types of objects** that the pointer visitor points to, **not** the type of the pointer itself.

- The meetvisitor method of Rector does not "know" which print method to call at compile time.
 - The type of the pointer visitor is Professor (Base). It is fixed.
 - The types of objects that the pointer visitor points to can change at runtime.
- Based on the contents of the pointer, different functions are called.

If visitor = &prof1 then Professor::print()

If visitor = &dean1 then Dean::print()

- Runtime polymorphism provides flexibility in design, as we will cover in this chapter.

Using a reference to base class to pass arguments

- Note that, in C++, we prefer to use references instead of pointers to pass arguments to functions.

Example:

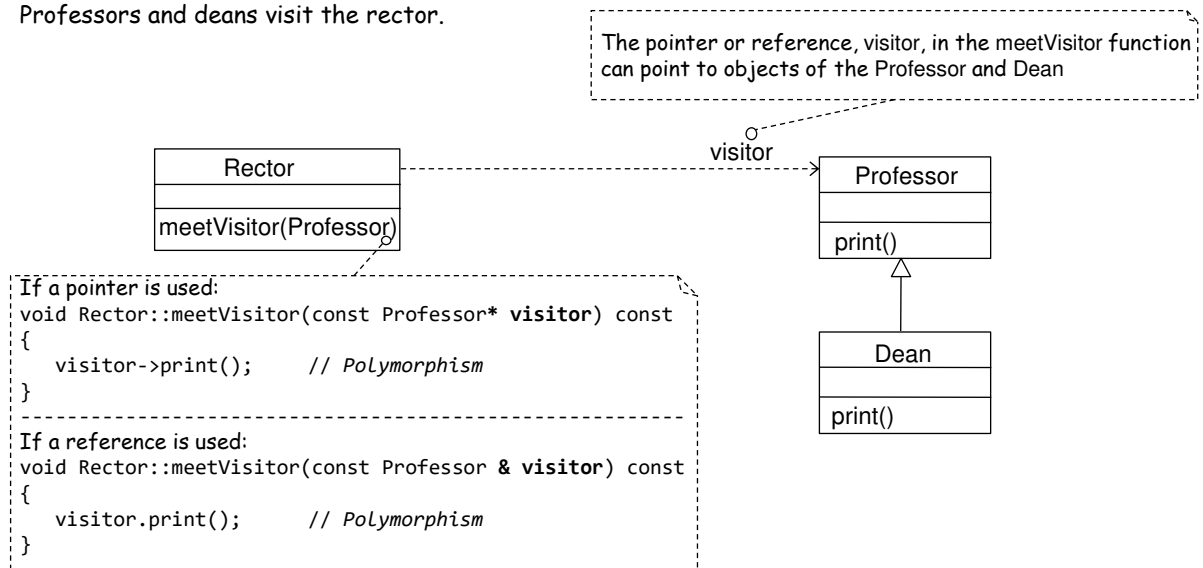
- We can write the meetVisitor method of the Rector class and the main function as follows:

```
// The input parameter is a reference to the Professor (Base) class
void Rector::meetVisitor(const Professor& visitor) const
{
    visitor.print();           // Polymorphism if print() is virtual
}

int main() {
    Rector itu_rector;
    Professor prof1("Professor 1", "Robotics");
    Dean dean1("Dean 1", "Computer Networks", "Engineering Faculty");
    char input_char;
    std::print("Professor or Dean (p/d)"); std::cin >> input_char;
    if (input_char == 'p') itu_rector.meetVisitor(prof1);
    if (input_char == 'd') itu_rector.meetVisitor(dean1);
    :
```

Example e08_1c.cpp

UML Class diagram of the design: Professors and deans visit the rector.



<http://akademi.itu.edu.tr/en/buzluca>
<http://www.buzluca.info>

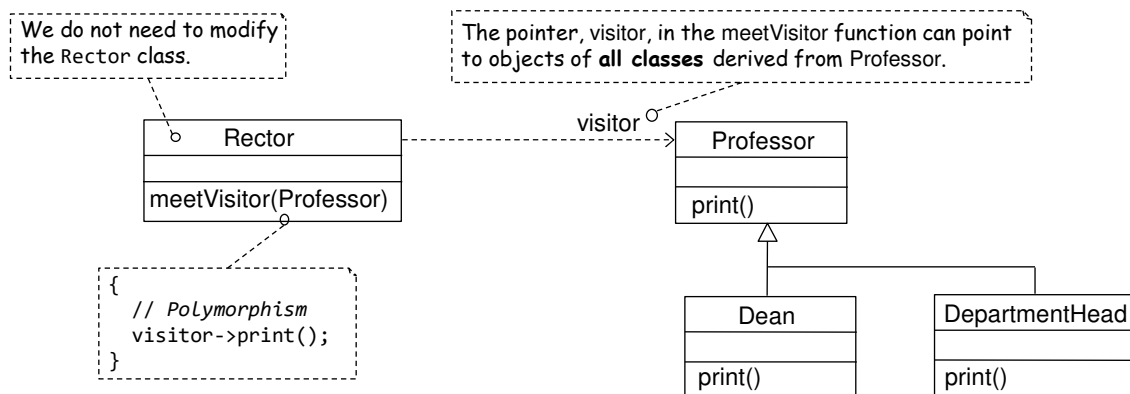


1999 - 2025 Feza BUZLUCA

8.11

Adding a new Professor type, e.g., department head, to the system:

- We often want to extend software systems by adding new features.
- While doing this, we do not want to modify the current system because this would increase costs.
- Since we designed the Rector class according to the Professor class, following the principle "**Design to Interface**", we can add a new Professor type, e.g., Department Head, without modifying the existing system (**Open-Closed** "open to extension, closed to modification" **Principle**).



<http://akademi.itu.edu.tr/en/buzluca>
<http://www.buzluca.info>



1999 - 2025 Feza BUZLUCA

8.12

Benefits of Polymorphism so far:

- The major advantage of polymorphism is flexibility.
- In our example, the rector is unaware of the type of visitor.
They can talk to a professor and a dean the same way (print()).
We do not need to insert a code into the Rector class to check the types of visitors.
- If we add a new professor type (a new class) to the system, for example, DepartmentHead, we do not need to change the Rector class.
- If a class derived from Professor is discarded from the system, we do not need to change Rector, either.
The input parameter of the meetVisitor method is a pointer or reference to the Professor class.
Therefore, we can call this method by sending either the address of a Professor object or the address of a Dean object.
So, this function can be applied to any class derived from the Professor.
- Polymorphism supports important design principles such as "Design to Interface" and "Open-Closed".

Early (static) binding vs. late (dynamic) binding**Type of the pointer and type of the pointed-to object:**

- A base-class pointer has two types associated with it: its **static type** and its **dynamic type**.
Static type is the type it was declared to point to (the compiler knows this), while **dynamic type** is the type of the object it is currently (at runtime) pointing to (or nullptr).

Example: Professor* visitor;

- The **static type** of the pointer, visitor, is a pointer to Professor (Professor*).
- The **dynamic type** of the pointer, visitor, varies according to the object it points to at runtime.
Remember, a base-class pointer can point to objects of all direct and indirect derived classes of that base.
 - When visitor is pointing to a Professor object, its dynamic type is a pointer to Professor.
 - When visitor is pointing to a Dean object, its dynamic type is a pointer to Dean.

Determining which function to call:

In our "Dean is a Professor" examples, there are two print() functions in memory, i.e.,
 Professor::print() and Dean::print().

How does the compiler know what function call to compile for the statement visitor->print(); ?

call Professor::print() or call Dean::print()

Early (static) binding:

- In e08_1a.cpp, without polymorphism (methods are **not virtual**), the compiler has no ambiguity about it.
- It considers the (static) type of the pointer, visitor, and always compiles a call to `Professor::print()`, regardless of the object type pointed to by the pointer or reference (dynamic type).
- In **early (static) binding**, the compiler matches the function call with the correct function definition at compile time.
- In C++, static binding is the default method of resolving function calls when the function is not a virtual function.
- Which function to call is determined at **compile time**.

Late (dynamic) binding:

- In e08_1b.cpp and e08_1c.cpp, since the print methods are virtual, the compiler does not “know” which function to call when compiling the program.
- The compiler cannot know it because **the decision is made at runtime**.
- So, instead of a simple function call, the compiler places a piece of code there.
- At runtime, when the function call is executed, the code that the compiler placed in the program finds out the type of the object whose address is in visitor and calls the appropriate `print()` function, i.e., `Professor::print()` or `Dean::print()`.
- Selecting a function at runtime is called **late binding** or **dynamic binding**.

How late binding (polymorphism) works: The virtual table

- The compiler creates a table—an array—of function addresses, called the **virtual table** for each class that has at least one virtual function.
- In examples e08_1b.cpp and e08_1c.cpp, the Professor and Dean classes each have their own virtual tables.
- Every virtual method in the class has an entry in the virtual table.

Example: Assume that the classes, Professor and Dean, contain two virtual functions.

```
class Professor{
public:
    virtual void readInfo();
    virtual void print() const;
private:
    std::string m_name;
    std::string m_researchArea;
};
```

Virtual Table of Professor

& Professor::readInfo
& Professor::print

```
class Dean : public Professor{
public:
    void readInfo();    // virtual
    void print() const; // virtual
private:
    std::string m_facultyName;
};
```

Virtual Table of Dean

& Dean::readInfo
& Dean::print

Calling virtual methods:

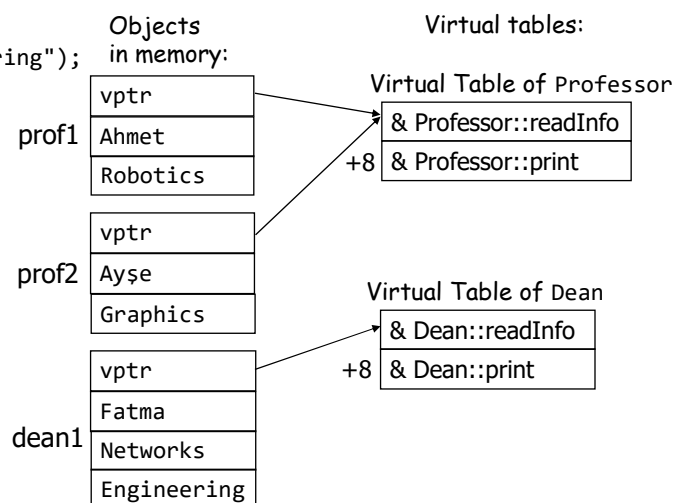
- For a statement that calls a virtual function, e.g, `visitor->print()`, the compiler does not (cannot) specify what function will be called at compile time.
 - For a virtual function call, the compiler creates code that will look at the active object's virtual table to get the address of the appropriate member function to run.
 - Thus, for virtual functions, the object itself (rather than the compiler) determines what function is called at runtime.
 - Objects of classes with virtual functions contain a pointer (vptr) to the class's virtual table.
 - The pointer, vptr, is used to access the object's virtual table at runtime.
- These objects are slightly larger than objects without virtual methods.

Calling virtual methods (cont'd):

Example:

```
int main(){
    Professor prof1("Ahmet", "Robotics");
    Professor prof2("Ayşe", "Graphics");
    Dean dean1("Fatma", "Networks", "Engineering");
    :
```

The objects of Professor and Dean contain pointers to their virtual tables.



Calling nonvirtual and virtual methods:**Nonvirtual print() function:**

- If the print() function was not virtual, the statement visitor->print() in the meetVisitor() method would be compiled as follows:

```
this ← visitor          ; this points to the active object
call Professor::print ; static binding, compile time
```

Virtual print() function (dynamic polymorphism):

- If the print() function is virtual, the statement visitor->print() in the meetVisitor() method will be compiled as follows:

```
this ← visitor          ; this points to the active object
ptr ← [this]            ; Read vptr from the object. ptr ← vptr
call [ptr + 8] <-----; dynamic binding, run-time
```

ptr points to the first row of the virtual table.

The first rows of the tables store the addresses of the readInfo() methods.

If the address length is 8 bytes in our system, we add 8 to the pointer to access the second row that stores the address of the print() method.

- Late binding requires a small amount of overhead but provides an enormous increase in power and flexibility.
- A few additional bytes per object and slightly slower function calls are small prices to pay for the power and flexibility offered by polymorphism.

Dynamic Polymorphism does not work with objects!

- Be aware that the dynamic polymorphism works only **with pointers and references** to objects, **not with objects** themselves.
- When we use an object's name to call a method, it is clear at compile time which method will be invoked.
- There is no need to determine which function to call at runtime.
- Thus, dynamic polymorphism does not work when we use an object's name to call a method.

Example:

```
int main(){
    Professor prof1("Ahmet", "Robotics");
    Professor prof2("Ayşe", "Graphics");
    Dean dean1("Fatma", "Networks", "Engineering");
    prof1.print();      // not polymorphic. Professor::print()
    dean1.print();     // not polymorphic. Dean::print()
```

- Calling virtual functions has an overhead because of indirect calls via tables.
- Do not declare functions as virtual if it is not necessary.

The rules about virtual functions

- To create a virtual (polymorphic) function in a derived class, its definition must have the **same signature** as the virtual function in the base class.
- Note that const specifications must also be identical. For example, if the base class method is const, the derived class method must also be const.
- If the signatures (parameters or const specifiers) of methods are different, the program will **compile without errors**, but **polymorphism** (virtual function mechanism) **will not work**.
- In this case, the function in the derived class redefines the function in the base (name hiding) and operates with static binding.

Example:

```
class Professor{
public:
    virtual void print() const;
};

class Dean : public Professor{
public:
    void print(); // Not virtual
};
```

Different signatures!

- No compiler error
- No dynamic polymorphism

You can try it by deleting the const specifiers of the print function of the Dean class in the programs e08_1b.cpp and e08_1c.cpp.

The rules about virtual functions (cont'd)

- If the signature (name, parameter list, and const specifier) of a function in a derived class is the same as that of the virtual function declared in the base class, then their **return types** must also be the same.

Otherwise, the derived class function will not compile.

Therefore, the program on the right will cause a compiler error.

The signatures (including const specifiers) **are different, return type is also different:**

- This is name hiding, and the new function will operate with static binding.
 - No compiler error
 - Name hiding
 - Static binding
 - **No dynamic polymorphism**

Example (same signatures, different return types):

```
class Professor{
public:
    virtual void print() const;
};

class Dean : public Professor{
public:
    int print() const; // Error!
};
```

Error: Same signatures but different return types

Example (different signatures, different return types):

```
class Professor{
    virtual void print() const;
};

class Dean : public Professor{
    int print(int) const; //OK! Name hiding
```

override Specifier

- Remember, to provide polymorphic behavior, the signatures (parameters or const specifiers) of virtual methods in base and derived classes must be the same.
- Otherwise, the program will compile without errors, but polymorphism (virtual function mechanism) will not work.
- However, it is easy to make a mistake (a typo) when specifying a virtual function in a derived class. For example, if we define a void `Print()` const method in the Dean class, it will not be virtual because the name of the corresponding method in the Professor class is different, i.e., void `print()` const.

The program may still be compiled and executed but may not work as expected.

- Similarly, the same thing will happen if we forget the const specifier in the derived class method.
- It is difficult to detect these kinds of errors.
- To avoid such errors, we can use the **override** specifier for every virtual function declaration in a derived class.

override Specifier (cont'd)**Example:**

```
class Professor{
public:
    virtual void print() const;
    virtual void readInfo();
    :
};

class Dean : public Professor{
public:
    void print() const override; // OK. Same signature
    void readInfo() override; // ERROR!
    :
};
```

Note: In the original image, a dashed box labeled 'Typo' points to the 'Info' in 'readInfo()' of the Professor class, and a dashed box labeled 'override;' points to the 'override;' in the Dean class.

- The override specification makes the compiler verify that the base class declares a virtual method with the same signature.
- If the base class does not have a virtual method with the same signature, the compiler generates an error.
- The override specification, like the virtual one, only appears within the class definition. It must not be applied to a method's definition (body).

- Always add an override specification to the declaration of a virtual function override.
- This guarantees that you have not made any mistakes in the function signatures.
- It safeguards you and your team from forgetting to change any existing function overrides when the signature of the base class function changes.

final Specifier

- Sometimes, we may want to prevent a method from being overridden in a derived class.
- We might want to limit how a derived class can modify the behavior of the base class interface, for example.
- We can do this by specifying that a function is final.

Example:

```
class Point {                      // Base Class (parent)
public:
    bool move(int, int) final;    // This method cannot be overridden
    :
};
```

Attempts to override move(int, int) in classes that have Point as a base will result in a compiler error.

final Specifier (cont'd)

- We can also specify an entire class as final.

Example:

```
class Parent final {              The Parent class cannot be used as a base class.
...
};

class Child : public Parent {      Compiler Error !
...
};
```

Example:

```
class ColoredPoint final : public Point {
:
};
```

Now, the compiler will not allow ColoredPoint to be used as a base class.

No further derivation is possible from the ColoredPoint class.

Summary: Overloading, Name Hiding, Overriding/Polymorphism**Overloading:**

- Remember, overloading occurs when two or more methods of the same class or multiple nonmember functions in the same namespace have the **same name but different parameters**.
- Overloaded functions operate with **static binding**.
- Which function to call is determined at **compile time**.
- Depending on the type of parameters, different functions are called.
- It is also called **static** polymorphism or ad hoc polymorphism.

Name hiding:

- Name hiding occurs when a derived class redefines the methods of the base class.
- The overridden methods may have the same or different signatures, but they will have different bodies.
- The methods **are not virtual**.
- Redefined methods operate with **static binding**.
- Which function to call is determined at **compile time**.

Summary: Overloading, Name Hiding, Overriding/Polymorphism (cont'd)**Dynamic polymorphism (Overriding):**

- The overridden methods have signatures identical to the base class's corresponding methods.
- The methods are specified as **virtual**.
- Overridden virtual methods operate with **dynamic binding**.
- Which function to call is determined at **runtime**.
- It is also called **dynamic** polymorphism, subtyping, or inclusion polymorphism.

A heterogeneous linked list of objects with polymorphism

- Remember: In example e07_19.zip, we developed a heterogeneous linked list that can contain Point and ColoredPoint objects.
- We will extend this program by adding virtual (polymorphic) print methods to the Point and ColoredPoint classes.

```
class Point {
public:
    virtual void print() const;    // virtual method
    :

class ColoredPoint : public Point {
public:
    void print() const override;  // virtual method
    :
```

We do not need to modify the Node class.

A heterogeneous linked list of objects with polymorphism (cont'd)

- We add a new method, printAll(), to the PointList class that iterates over the list and calls print() methods of all elements consecutively.
- Since some elements are Point objects and some are ColoredPoint objects, different print() methods will be invoked depending on the type of the elements.

```
void PointList::printAll() const {
    if (m_head)                                // if the list is not empty
    {
        Node* tempPtr{ m_head };                // A pointer points to the first node of the list
        while (tempPtr) {
            tempPtr->getPoint()->print();        // POLYMORPHISM
            tempPtr = tempPtr->getNext();         // go to the next node
        }
    }
    else std::println("The list is empty");
}
```

Get the address of the object from the current node.

Call the print() pointed by the pointer received from the current node.

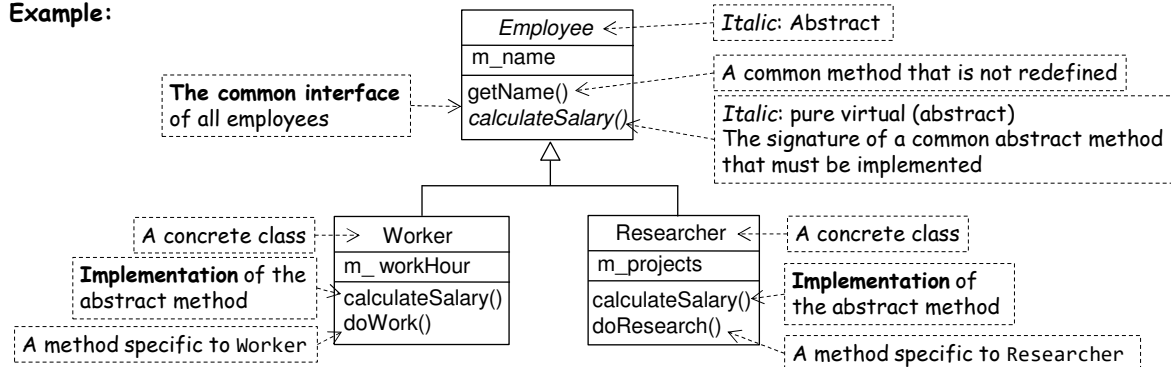
Example e08_2.zip

Remember, there is a `std::list` class in the C++ Standard Library. You do not need to write a class to define linked lists.

Abstract Classes

- Sometimes, we do not need to create base class objects; we only need derived ones.
- The base class exists only to store the common properties of derived classes and to present their common services (responsibilities).

Example:



- In this system, we will never have generic Employee objects.
- We will create either Worker or Researcher objects.
- This kind of base class (e.g., Employee) is called an **abstract class**, meaning no actual objects will be created from it.

Abstract Classes (cont'd)

Pure virtual functions:

- When we decide to create an abstract base class, we can instruct the compiler to prevent any class user from ever making an object of that class.

This would give us more freedom in designing the base class because we would not need to plan for actual objects of the class but only for data and functions that derived classes would use.

- To tell the compiler that a class is abstract, we define at least one pure virtual function in that class.
- A pure virtual function is a virtual function without a body.

The body of the virtual function in the base class is removed, and the notation `=0` is added to the function declaration.

Example:

- The Employee class is abstract, and the method `calculateSalary()` is a **pure virtual function**.

```

class Employee{           // Abstract! It is not possible to create objects
public:
    virtual double calculateSalary() const = 0;           // pure virtual function

int main(){
    Employee employeeObject{"Employee 1"};               // Compiler Error!
    employeePtr = new Employee {"Employee 1"};           // Compiler Error!
  
```


Example: Employee, worker, and researcher. Employee is an **abstract** class

```
class Employee{           // Abstract! It is not possible to create objects
public:
    Employee::Employee(const std::string& in_name) : m_name{ in_name }    // constructor
    {}
    const std::string& getName() const;                                   // 1. Not virtual. A common method
    virtual void print() const;                                          // 2. Virtual (but not abstract)
    virtual double calculateSalary() const = 0;                          // 3. Pure virtual (abstract)
private:
    std::string m_name;
};

void Employee::print() const                                           // The body of the virtual function
{
    std::println("Name: {}", m_name);
}
```

- The calculateSalary() method is not defined (implemented) in the Employee class.
- It is an **abstract** (pure virtual) method.

Creating instances (objects) of an abstract class is not possible.

Example: Employee is an **abstract** class (cont'd)

- The Employee class is an incomplete description of an object because the calculateSalary() function is not defined (it does not have a body).
Therefore, it is abstract, and we are not allowed to create instances (objects) of the Employee class.
- This class exists solely for the purpose of deriving classes from it.

```
Employee employeeObject{"Employee 1"};    // Compiler Error!
Employee * employeePtr;                   // OK. Pointer is not an object
employeePtr = new Employee {"Employee 1"}; // Compiler Error!
```

- Since you cannot create its objects, you cannot pass an Employee **by value** to a function or return an Employee **by value** from a function.

The derived classes specify how each pure virtual function is implemented:

Example: Employee is an **abstract** class (cont'd)

- The Employee class determines the signatures (interfaces) of the virtual functions.
- The creators of the derived classes (e.g., Worker and Researcher) specify how each pure virtual function is implemented.
- Classes derived from the Employee class will define (implement) the calculateSalary() function.
- If a pure virtual function of an abstract base class is not defined in a derived class, then the pure virtual function will be inherited as is, and the derived class will also be an abstract class.
- Classes without pure virtual methods are called **concrete classes**.

Example (cont'd): Employee, Worker, and Researcher

```
class Worker : public Employee{
public:
    void print() const override;           // Redefined print function
    double calculateSalary() const override; // Concrete virtual function
    :
};

void Worker::print() const                // Redefined virtual function
{
    Employee::print();
    std::println("I am a worker");
    std::println( "My work Hours per month: {}", m_workHour);
}

// Concrete (implemented) virtual function
double Worker::calculateSalary() const
{
    return 1000* m_workHour;              // 1000TL per hour
}
```

We can similarly derive a Researcher class from Employee.

Example (cont'd): Employee, Worker, and Researcher

```

int main(){
    // Employee employee1{"Employee 1"};           // Error! Employee abstract
    Employee * employeePtr;                         // OK. Pointer, not an object
    // employeePtr = new Employee {"Employee 1"};   // Error!

    Employee* arrayOfEmployee[5]{};                // An array of 5 pointers to Employee
    Worker worker1{ "Worker 1", 160 };             // Work hours per month = 160
    arrayOfEmployee[0] = &worker1;                 // Address of worker1 to the array
    std::println( arrayOfEmployee[0]->getName() );  // OK! common function

    Researcher researcher1{ "Researcher 1", 1 };     // The number of projects = 1
    arrayOfEmployee[1] = &researcher1;              // Addr. of researcher1 to the array
    :
    for (unsigned int i = 0; i < 5; i++) {
        arrayOfEmployee[i]->print();                // polymorphic method calls
        std::println("Salary = {}", arrayOfEmployee[i]->calculateSalary() );
    }
    return 0;
}

```

Example e08_3.cpp

A design principle: "Design to an interface, not an implementation"

- Software design principles are guidelines (best practices) offered by experienced practitioners in the design field.
- "Design to an interface, not an implementation" is a principle that helps us design flexible systems that can handle changes.
- Here, the **interface** refers to the signatures of the common services (behaviors) provided by different classes.

For example, Workers and Researchers can both calculate their salaries and print their information.

- The **implementation** refers to how different classes define (implement) common services (or behaviors). For example, the Worker class has a unique method of calculating its salary.

The Researcher class can also calculate the salary but in another way.

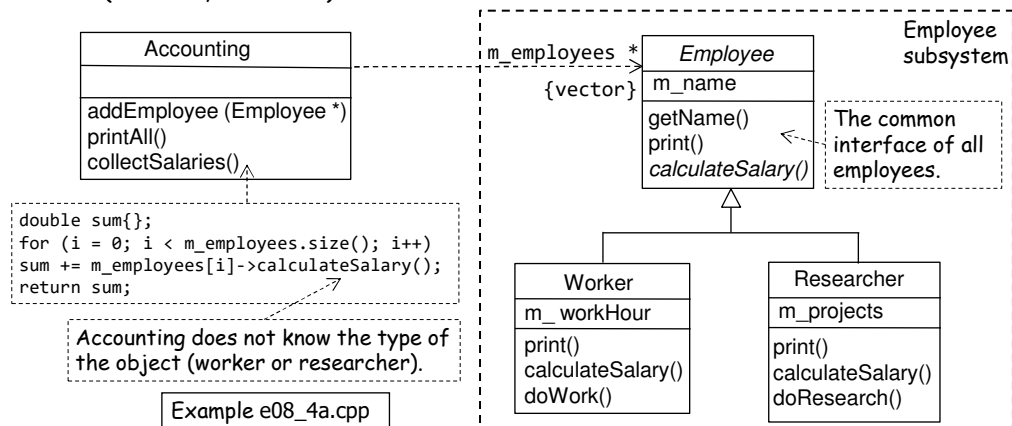
The interfaces of some services are the same, but their implementations are different.

For example, the signature (interface) of the virtual calculateSalary() function is the same for both Workers and Researchers.

However, the implementation (body) of this method is different in these classes.

A design principle: "Design to an interface, not an implementation" (cont'd)**Example:** Workers, researchers, and the accounting system in a company

- We need to design an accounting system that performs financial operations related to workers and researchers.
- We will design the Accounting class according to the base class Employee that presents the common interface (services, behaviors) of workers and researchers.



<http://akademi.itu.edu.tr/en/buzluca>
<http://www.buzluca.info>

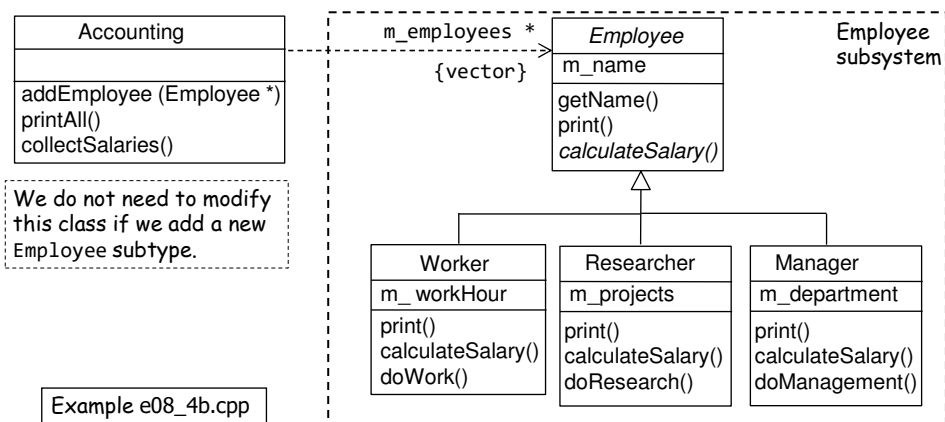


1999 - 2025 Feza BUZLUCA

8.39

Example: Workers, researchers, and the accounting system in a company

- We designed the Accounting class according to a general Employee type (interface).
- Accounting is not aware of the concrete types (implementations) (Worker and Researcher).
- If we need to add a new type of employee, e.g., a Manager, to our system, we do not need to change the Accounting class.



<http://akademi.itu.edu.tr/en/buzluca>
<http://www.buzluca.info>



1999 - 2025 Feza BUZLUCA

8.40

The Open-Closed Principle

"Software entities (classes, modules, functions, etc.) should be **open for extension** but **closed for modification**".

- We should strive to write code that does not have to be changed every time the requirements change or new functionalities are added to the system.
- We should create flexible designs to take on new functionality to meet changing requirements without modifying the existing code.

The polymorphism concept in OOP and the principles "*Find what varies and encapsulate it*" and "*Design to interface not to an implementation*" support the "*Open-Closed Principle*".

For example, we can add a new type of employee, such as a Manager, to our system without changing the existing code.

Virtual Constructors?

Can constructors be virtual?

No, constructors cannot be virtual.

- When creating an object, we usually already know what kind of object we are creating and can specify this to the compiler.
- Thus, there is no need for virtual constructors.
- Also, an object's constructor sets up its virtual mechanism (the virtual table) in the first place.
- Of course, we do not see the source code for this, just as we do not see the code that allocates memory for an object.
- Virtual functions cannot even exist until the constructor has finished its job, so **constructors cannot be virtual**.

Virtual Destructors

- Recall that a derived class object typically contains data from both the base and derived classes.
- To ensure that such data is properly disposed of, it may be essential for destructors for both base and derived classes are called.

Example:

Example e08_5a.cpp

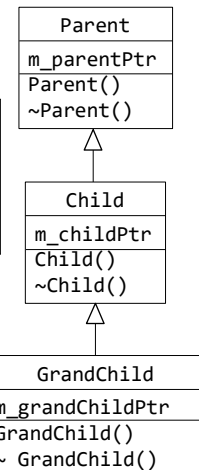
- The classes Parent, Child, and GrandChild contain pointers to integer arrays.
- Constructors allocate memory for arrays, while destructors release that memory.

```
Parent() {
    m_parentPtr = new int[10];
}
~Parent() {
    delete[] m_parentPtr;
}
```

```
Child() {
    m_childPtr = new int[10];
}
~Child() {
    delete[] m_childPtr;
}
```

```
int main() {
    GrandChild grandchild_object;
    std::println("..Program terminates..");
    return 0;
}
```

The Output: Parent constructor
 Child constructor
 GrandChild constructor
 ..Program terminates..
 GrandChild destructor
 Child destructor
 Parent destructor

**Virtual Destructors (cont'd)**

- In the previous example, destructors are **not virtual**. We defined an **automatic object** of the GrandChild class. Constructors and destructors were called in the correct order.
- When we create a **dynamic object** of the GrandChild class pointed to by a pointer to the Parent class, what happens if this object is deleted?

Example:

```
int main(){
    Parent* parentPtr{};
    parentPtr = new GrandChild;
    std::println("-----");
    delete parentPtr;
    std::println("..Program terminates..");
    return 0;
}
```

Example e08_5b.cpp

The Output:

```

>Parent constructor
>Child constructor
>GrandChild constructor
-----
>Parent destructor
..Program terminates..

```

- In this example, parentPtr points to an object of the GrandChild class, but **only the Parent class destructor is called** while deleting the pointer. Child and GrandChild destructors are not called. We encountered the same problem when we previously called nonvirtual functions using a base pointer.
- Remember: If a function is not virtual, only the base class version will be called when it is invoked using a base class pointer, even if the pointer's content is the address of a derived class object (**static binding**). Thus, Child and GrandChild destructors are never called. This could be a problem if these destructors did something important.

Virtual Destructors (cont'd)

- To ensure that the destructors of derived classes are called **for dynamic objects**, we need to specify **destructors as virtual**.
- To implement a virtual destructor in a derived class, we just add the keyword `virtual` to the destructor declaration in the base class.

This makes the destructors in every class derived from the base class `virtual`.

- The virtual destructor calls through a pointer or a reference have **dynamic binding**, so the called destructor will be selected at runtime.
- To fix the problem in example `e08_5b.cpp`, we add the `virtual` keyword to the destructor declaration in the Parent class.

```
class Parent{
public:
    Parent();
    virtual ~Parent(); // virtual destructor
};

int main(){
    :
    delete parentPtr;
```

Example `e08_5c.cpp`

The Output:

```
Parent constructor
Child constructor
GrandChild constructor
-----
->GrandChild destructor
->Child destructor
->Parent destructor
..Program terminates..
```