

POLYMORPHISM

There are three major concepts in object-oriented programming:

1. Encapsulation (Classes, Objects)
   Data and related functions are placed into the same entity.
   Data abstraction
   Information hiding (`public:` interface, `private:` implementation)
2. Inheritance
   Is-a relation, generalization-specialization, reusability
   Common interfaces
3. Polymorphism (dynamic, subtyping)
   The run-time decision for function calls (dynamic method binding)
   Overriding of methods
   Needs inheritance
   Improves the design with common interfaces

What we refer to in this lecture slides as “polymorphism” is formally known as **dynamic**, subtyping (or inclusion) polymorphism.

Polymorphism in real life:

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

Example: The Dean is a professor.

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 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** message `print()`.
- 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.

Polymorphism means “**taking many shapes**”.

The rector’s single instruction is polymorphic because it works differently for different kinds of academic staff.

Polymorphism (dynamic) in programming:

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

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;  // Can point to Professor and Dean objects
  ...        //The address pointed to by ptr will be determined in runtime
  ptr = &professor_obj;
   or
  ptr = &dean_obj;

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

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

Calling redefined, nonvirtual member functions using pointers (name hiding)

The first example shows what happens when a base class and derived classes have functions with the same signature (name and parameters) accessed using pointers. In this example, the functions are **not virtual** (**no polymorphism**).

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 (contd)

Example (contd): Professors and deans visit the rector

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

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

A pointer to the base class

Since the input parameter is a pointer to the `Professor` (base) class, we can call this method 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 (contd)

Example (contd): Professors and deans visit the rector

See Example e08_1a.cpp

```
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 c;
   std::print("Professor or Dean (p/else)); std::cin >> c;
   if (c=='p') ptr = &prof1;         // ptr points to a professor
       else    ptr = &dean1;          // ptr points to a dean
   itu_rector.meetVisitor(ptr);       // which print?
```

In this example, at the statement `visitor->print()`, the `print()` function of the base class (`Professor`) is 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. `Professor *visitor;`

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

Calling redefined, virtual member functions using pointers (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.

```
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
{                               // We did not change the methods of Rector
   visitor->print();       // which print?
}
```

See Example e08_1b.cpp

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

- Since `print()` functions are virtual, **different functions** are executed depending on the **contents** of the pointer, **not** on 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 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.

If `visitor = &prof1` then `Professor::print()`
If `visitor = &dean1` then `Dean::print()`

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.

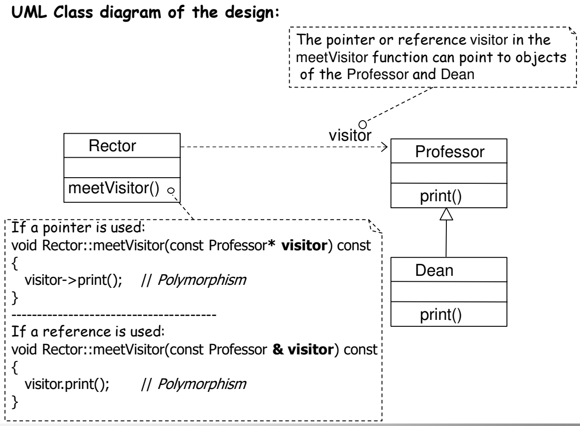
We can write the `meetVisitor` method of the `Rector` class and the main function as follows:

```
// The input parameter is a reference to 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 c;
   std::print("Professor or Dean (p/d)"); std::cin >> c;
   if (c == 'p') itu_rector.meetVisitor(prof1);
   if (c == 'd') itu_rector.meetVisitor(dean1);
    :
```

See Example e08_1c.cpp

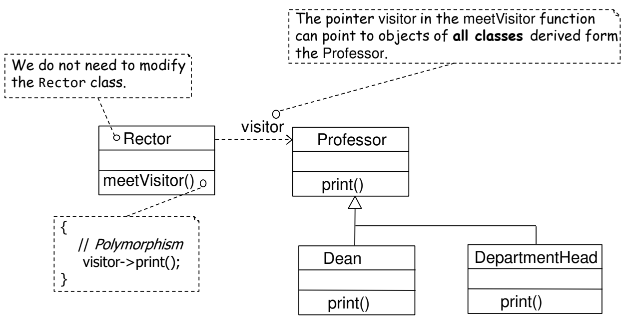
UML Class diagram of the design:



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.
- It is also true if a class derived from the `Professor` is discarded from the system.
  The input parameter of the `meetVisitor` method is a pointer or reference to the `Professor` class.
  Therefore, we can call this method by sending both an address of a `Professor` object and an address of a `Dean` object.
  So, this function can be applied to any class derived from the `Professor`.
- In the following slides, we will cover important design principles supported by polymorphism, i.e., "Design to Interface" and "Open-Closed Principle."

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



Early (static) binding vs late (dynamic) binding

Type of the pointer and type of the pointed-to object:

- A pointer to a base class has two types, i.e., static type and dynamic type.
  Example: `Professor* visitor;`
- The ***static type*** of the pointer `visitor` is a pointer to `Professor` (`Professor*`).
- Since `visitor` is a pointer to a base class, it also has a ***dynamic*** *type*, which varies according to the object it points to.
  Remember, a pointer to a base class can point to objects of all direct and indirect derived classes from 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 visitor->print(); ?

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


Early (static) binding:

In e08_1a.cpp, without polymorphism, 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).

- Connecting to functions during compilation is called ***early*** (***static***) ***binding***.
- Binding means connecting the function call to the function.
- Static binding is the standard operating method for the compilers.
- Which function to call is determined at **compile-time**.

Late (dynamic) binding:

In e08_1b.cpp and e08_1c.cpp, 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

- Remember: For a regular object **without any virtual methods**, only its data are stored in memory.
- When a member function is called for such an object, the address of the object is available in **this** pointer, which the member function uses (usually invisibly) to access the object's data.
- Every time a member function is called, the compiler assigns the address of the object for which the function is called to `this` pointer (see slide 4.32).

Calling virtual methods:

- When a derived class with virtual functions is specified, the compiler creates a table—an array—of function addresses called the **virtual table**.
  In the examples e081a.cpp and e081b.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.
- Objects of classes with virtual functions contain a pointer (**vptr**) to the class's virtual table.
  These objects are slightly larger than objects without virtual methods.

Calling virtual methods, the virtual table:

When a virtual function is called for an object, instead of specifying what function will be called in compile-time, the compiler creates a code that will look at the object's virtual table to get the address of the appropriate member function to run.

Thus, for virtual functions, the object itself determines what function is called at runtime rather than the compiler.

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;
};
```

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

Virtual Table of Profesor

& Professor::readInfo
& Professor::print

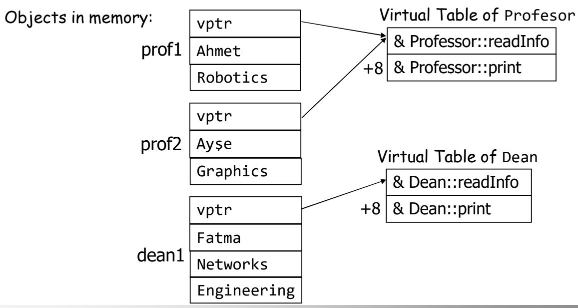
Virtual Table of Dean

& Dean::readInfo
& Dean::print

Calling virtual methods, the virtual table (contd):

The objects of the Professor and Dean will contain a pointer to their virtual tables.

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



Calling virtual methods, the virtual table (contd):

Nonvirtual `print()` function:

If the `print()` function was <u>not virtual</u>, 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 (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.

Polymorphism (dynamic) 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.

```
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`.

Example:

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

- If the signatures (parameters or const specifiers) of methods are different, the program will **compile without errors**, but the **polymorphism** (`virtual` function mechanism) **will not work**.
- The function in the derived class redefines the function in the base (name hiding), as we covered in Chapter 7.
- This new function will, therefore, operate with static binding as in program e08_1a.cpp.
- You can try it by deleting `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 (contd)

- The return type of a virtual function in a derived class must be the same as that in the base class.

Example:

```
class Professor{
public:
  virtual void print() const;
    :
};      Error: Same signatures but
        different return types
class Dean : public Professor{
public:
  int print() const; // Error!
    :
```

- If the function name, parameter list, and const specifier of a function in a derived class are the same as those of a `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 left will cause a compiler error.
- A different return type will not cause a compiler error if the signatures or const specifiers are already different.

  This is (name hiding); the new function will operate with static binding.

Example:

```
class Professor{                          class Dean : public Professor{
  virtual void print() const;               int print(int) const; //OK! Compile-time
```

Different signatures: Name hiding. **No compiler error**. **No polymorphism**. Static binding

`override` Specifier

- Remember, to provide a 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 the 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 be as expected.
- Similarly, the same thing will happen if we forget to `const` specifier in the derived class.
- 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 (contd)

Example:

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

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

- 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.

It happens if we want to limit how a derived class can modify the behavior of the base class interface.

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 with `Point` as a base will result in a compiler error.

`final` Specifier (contd)

We can also specify an entire class as `final`.

Example:

```
class Parent final {
 ...
};

class Child : public Parent {
 ...
};
```

The `Parent` class cannot be used as a base class.

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 from the `ColoredPoint` class is possible.

Overloading, Name Hiding, Overriding/Polymorphism

Overloading:

- Remember, overloading occurs when two or more methods of the <u>same class</u> or multiple nonmember methods 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 the parameters, different functions are called.
- It is also called **static**, ad hoc polymorphism.

Name hiding:

- Name hiding (compile-time overriding) occurs when a derived class redefines the methods of the base class.
- The overridden methods may have <u>the same or different signatures</u>, 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**.

Overloading, Name Hiding, Overriding/Polymorphism (contd)

Polymorphism (Overriding):

- The overridden methods have <u>identical signatures to the base class's corresponding methods</u>.
- 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**, subtyping polymorphism.

Types of Polymorphism:

- 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 (parameters).
- In this class (usually in OOP), **polymorphism** refers to **virtual functions** that provide **dynamic (run-time)** polymorphism through an interface provided by a base class (subtyping or inclusion polymorphism).
- Other types of polymorphism:
  - **Static** polymorphism
    - Ad hoc polymorphism: function and operator overloading
      For example, `int i = add(5);` or `Point pt = add(point_obj);`
    - Parametric polymorphism: function and class templates
      Generic programming (see Chapter 09)

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 (contd)

- 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.

See Example e08_2.zip

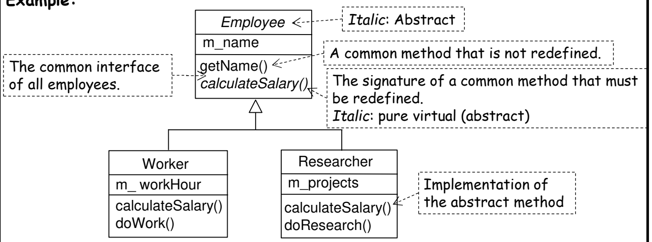
Remember, there is a **`std::list`** class in the standard library of C++. You do not need to write a class to define linked lists.

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 (contd)

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 <u>at least one</u> ***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
```

Each derived class will (and must) implement the body of this method.

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; // A common method, not redefined
 virtual void print() const;                    // virtual (not abstract)
 virtual double calculateSalary() const = 0; // pure virtual function
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

- 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"};  // Compiler Error!
Employee * employeePtr = new Employee {"Employee 1"};   // 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

- The `Employee` class determines the signatures (interfaces) of the virtual functions.
- The authors of the derived classes (e.g., `Worker and Researcher`) specify how each pure virtual function is implemented.
- Any class derived from the `Employee` class must define (implement) the `calculateSalary()` function.

  If it does not, then it is also an abstract class.
- 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.

Example (contd): 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();
  cout << " I am a worker" << endl;
  cout << "My work Hours per month: " << m_workHour << endl;
}

// Concrete (implemented) virtual function
double Worker::calculateSalary() const
{
    return 105* m_workHour;       // 105TL per hour
}
```

We can similarly derive a `Researcher` class from the `Employee`.

Example (contd): Employee, worker, and researcher

```
int main(){
  // Employee employee1{"Employee 1"};    // Error! Employee abstract
  // Employee * 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;     // Addr. of the worker1 to the array
  std::println( arrayOfEmployee[0]->getName() );  // OK! common function

  Researcher researcher1{ "Researcher 1", 1 };    // #projects = 1
  arrayOfEmployee[1] = &researcher1;  // Addr. 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;
}
```

See 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 to design flexible systems that can handle changes.
- Here, **the interface** refers to the signatures of the common services (behaviors) given by different classes.

  For example, `Workers` and `Resarchers` 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 `Resarchers`.

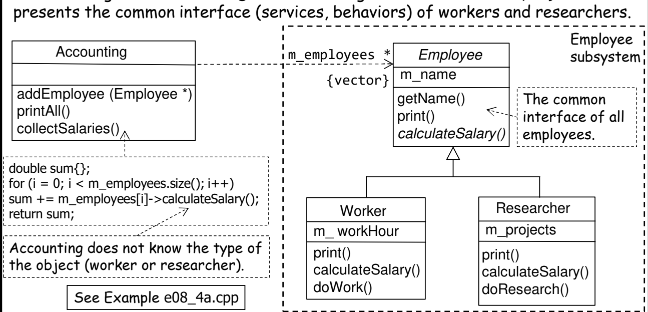
  However, the implementation (body) of this method is different in these classes.

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

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.

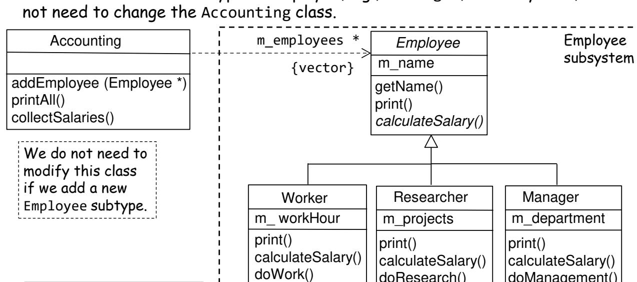
We will design the `Accounting` class according to the base class `Employee` that presents the common interface (services, behaviors) of workers and researchers.



See Example e08_4a.cpp

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

- We designed the `Accounting` class according to a general `Employee` (interface) type.
- Accounting is not aware of the concrete types (`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.



See Example e08_4b.cpp

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 OOP concept polymorphism 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**.

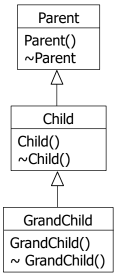
Virtual Destructors

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 that destructors for both base and derived classes are called.

Remember the example e07_8.cpp on slide 7.36.

Example:




```
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
```

See Example e08_5a.cpp

Virtual Destructors (contd)

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;
  return 0;
}
```

See Example e08_5b.cpp

The Output:

```
Parent constructor
Child constructor
GrandChild constructor
------------------
Parent destructor
```

- In this example, `parentPtr` points to an object of the `GrandChild` class, but only the `Parent` class destructor is called while deleting the pointer.

  We encountered the same problem when we previously called ordinary functions using a base pointer.
- 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 (contd)

- 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();
   :
};
```

See Example e08_5c.cpp

The Output:

```
Parent constructor
Child constructor
GrandChild constructor
------------------
GrandChild destructor
Child destructor
Parent destructor
```