Module 30: Programming in C++

Polymorphism: Part 5: Staff Salary Processing using C++

Intructors: Abir Das and Sourangshu Bhattacharya

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

{*abir, sourangshu*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in Modern C++

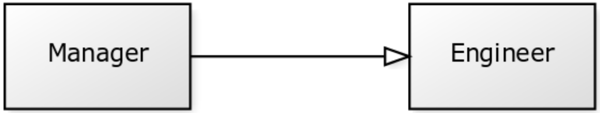
by **Prof. Partha Pratim Das**

Module Objectives

- Understand design with class hierarchy
- Understand the process of design refinement to get to a good solution from a starting one

Module Outline

C++ Solution: Non-Polymorphic Hierarchy: Engineer + Manager



- How to represent Engineers and Managers?
  - Non-Polymorphic `class` hierarchy
- How to initialize objects?
  - Constructor / Destructor
- How to have a collection of mixed objects?
  - `array` of base class pointers
- How to model variations in salary processing algorithms?
  - Member functions
- How to invoke the correct algorithm for a correct employee type?
  - Function switch
  - Function pointers

C++ Solution: Non-Polymorphic Hierarchy: Engineer + Manager

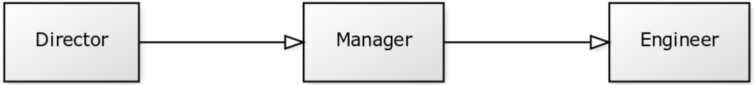
```cpp
#include <iostream>
#include <string>
using namespace std;

enum E_TYPE { Er, Mgr };

class Engineer {
protected:
    string name_; E_TYPE type_;
public:
    Engineer(const string& name, E_TYPE e = Er) : name_(name), type_(e) { }
    E_TYPE GetType() { return type_; }
    void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer {
    Engineer *reports_[10];
public:
    Manager(const string& name, E_TYPE e = Mgr) : Engineer(name, e) { }
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};
```

C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager

```
int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
    Manager m1("Kamala"), m2("Rajib");
    Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3 };

    for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
        E_TYPE t = staff[i]->GetType();
        if (t == Er)
            staff[i]->ProcessSalary();
        else if (t == Mgr)
                ((Manager *)staff[i])->ProcessSalary();
             else cout << "Invalid Staff Type" << endl;
    }
}
```

```
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
```

C++ Solution: Non-Polymorphic Hierarchy: Engineer + Manager + Director



- How to represent Engineers, Managers, and Directors?
  - Non-Polymorphic `class` hierarchy
- How to initialize objects?
  - Constructor / Destructor
- How to have a collection of mixed objects?
  - `array` of base class pointers
- How to model variations in salary processing algorithms?
  - Member functions
- How to invoke the correct algorithm for a correct employee type?
  - Function switch
  - Function pointers

C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager + Director

```
#include <iostream>
#include <string>
using namespace std;
enum E_TYPE { Er, Mgr, Dir };

class Engineer {
protected:
    string name_; E_TYPE type_;
public:
    Engineer(const string& name, E_TYPE e = Er) : name_(name), type_(e) {}
    E_TYPE GetType() { return type_; }
    void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};
class Manager : public Engineer {
    Engineer *reports_[10];
public:
    Manager(const string& name, E_TYPE e = Mgr) : Engineer(name, e) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};
class Director : public Manager {
    Manager *reports_[10];
public:
    Director(const string& name) : Manager(name, Dir) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};
```

C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager + Director

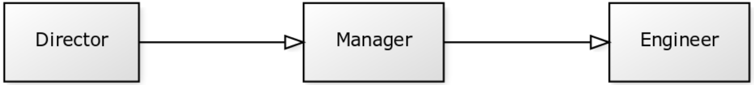
```
int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
    Manager m1("Kamala"), m2("Rajib");
    Director d("Ranjana");
    Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3, &d };

    for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
        E_TYPE t = staff[i]->GetType();
        if (t == Er)
            staff[i]->ProcessSalary();
        else if (t == Mgr)
                ((Manager *)staff[i])->ProcessSalary();
             else if (t == Dir)
                    ((Director *)staff[i])->ProcessSalary();
                  else cout << "Invalid Staff Type" << endl;
    }
}
```

```
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
Ranjana: Process Salary for Director
```

C++ Solution: Non-Polymorphic Hierarchy: Advantages and Disadvantages

- **Advantages**
  - Data is encapsulated
  - Hierarchy factors common data members
  - Constructor / Destructor to manage lifetime
  - `struct`-specific functions made member function (overridden)
  - `E_Type` subsumed in `class` – no need for `union`
  - Code reuse evidenced
- **Disadvantages**
  - Types of objects are managed explicitly by `E_Type`:
    - ▷ Difficult to extend the design – addition of a new type needs to:
      - – Add new type code to `enum E_Type`
      - – Application code need to have a new case (`if-else`) based on the new type
    - ▷ Error prone because the application programmer has to cast to right type to call `ProcessSalary`
- **Recommendation**
  - Use a polymorphic hierarchy with dynamic dispatch

C++ Solution: Polymorphic Hierarchy Engineer + Manager + Director



- How to represent Engineers, Managers, and Directors?
  - Polymorphic `class` hierarchy
- How to initialize objects?
  - Constructor / Destructor
- How to have a collection of mixed objects?
  - `array` of base class pointers
- How to model variations in salary processing algorithms?
  - Member functions
- How to invoke the correct algorithm for a correct employee type?
  - Virtual Functions

C++ Solution: Polymorphic Hierarchy Engineer + Manager + Director

```cpp
#include <iostream>
#include <string>
using namespace std;

class Engineer {
protected:
    string name_;
public:
    Engineer(const string& name) : name_(name) {}
    virtual void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer {
    Engineer *reports_[10];
public:
    Manager(const string& name) : Engineer(name) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};

class Director : public Manager {
    Manager *reports_[10];
public:
    Director(const string& name) : Manager(name) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};
```

C++ Solution: Polymorphic Hierarchy Engineer + Manager + Director

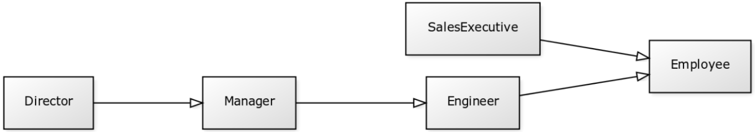
```
int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
    Manager m1("Kamala"), m2("Rajib");
    Director d("Ranjana");
    Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3, &d };

    for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i)
        staff[i]->ProcessSalary();
}
```

```
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
Ranjana: Process Salary for Director
```

C++ Solution: Polymorphic Hierarchy: Advantages and Disadvantages

- **Advantages**
  - Data is fully encapsulated
  - Polymorphic Hierarchy removes the need for explicit `E_Type`
  - Application code is independent of types in the system (`virtual` functions manage types through polymorphic dispatch)
  - High Code reuse – code is short and simple
- **Disadvantages**
  - Difficult to add an employee type that is not a part of this hierarchy (for example, employees of *Sales Division*
- **Recommendation**
  - Use an abstract base class for employees

C++ Solution: Polymorphic Hierarchy (Flexible) Engineer + Manager + Director + Others



- How to represent Engineers, Managers, Directors, etc.?
  - Polymorphic `class` hierarchy with an Abstract Base Employee
- How to initialize objects?
  - Constructor / Destructor
- How to have a collection of mixed objects?
  - `array` of base class pointers
- How to model variations in salary processing algorithms?
  - Member functions
- How to invoke the correct algorithm for a correct employee type?
  - Virtual Functions (Pure in Employee)

C++ Solution: Polymorphic Hierarchy (Flexible) Engineer + Manager + Director + Others

```
#include <iostream>
#include <string>
using namespace std;
class Employee {
protected: string name_;
public:
    virtual void ProcessSalary() = 0;
    virtual ~Employee() { }
};
class Engineer: public Employee { public:
    Engineer(const string& name) { name_ = name; }
    void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};
class Manager : public Engineer { Engineer *reports_[10]; public:
    Manager(const string& name) : Engineer(name) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};
class Director : public Manager { Manager *reports_[10]; public:
    Director(const string& name) : Manager(name) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};
class SalesExecutive : public Employee { public:
    SalesExecutive(const string& name) { name_ = name; }
    void ProcessSalary() { cout << name_ << ": Process Salary for Sales Executive" << endl; }
};
```

C++ Solution: Polymorphic Hierarchy (Flexible) Engineer + Manager + Director + Others

```
int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
    Manager m1("Kamala"), m2("Rajib");
    SalesExecutive s1("Hari"), s2("Bishnu");
    Director d("Ranjana");

    Employee *staff[] = { &e1, &m1, &m2, &e2, &s1, &e3, &d, &s2 };

    for (int i = 0; i < sizeof(staff) / sizeof(Employee*); ++i)
        staff[i]->ProcessSalary();
}
```

```
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Hari: Process Salary for Sales Executive
Shambhu: Process Salary for Engineer
Ranjana: Process Salary for Director
Bishnu: Process Salary for Sales Executive
```

C++ Solution: Polymorphic Hierarchy (Flexible): Advantages and Disadvantages

- **Advantages**
  - Data is fully encapsulated
  - Flexible Polymorphic Hierarchy makes addition of any class possible on the hierarchy
  - Application code is independent of types in the system (`virtual` functions manage types through polymorphic dispatch)
  - Maximum Code reuse – code is short and simple
- **Disadvantages**
  - Still needs to maintain employee objects in code and add them to the staff array - this is error prone
- **Recommendation**
  - Use vector as a collection and insert staff as created

C++ Solution: Polymorphic Hierarchy (Flexible) Engineer + Manager + Director + Others

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
class Employee { protected: string name_; // Name of the employee
    vector<Employee*> reports_;           // Collection of reportees aggregated
public: virtual void ProcessSalary() = 0; // Processing salary
    virtual ~Employee() { }
    static vector<Employee*> staffs;      // Collection of all staffs
    void AddStaff(Employee* e) { staffs.push_back(e); };  // Add a staff to collection
};
class Engineer : public Employee { public:
    Engineer(const string& name) { name_ = name;     // Why init like name_(name) won't work?
                                   AddStaff(this); } // Add the staff
    void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};
class Manager : public Engineer { public: Manager(const string& name) : Engineer(name) { }
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};
class Director : public Manager { public: Director(const string& name) : Manager(name) { }
    void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};
class SalesExecutive : public Employee { public:
    SalesExecutive(const string& name) { name_ = name; AddStaff(this); }   // Add the staff
    void ProcessSalary() { cout << name_ << ": Process Salary for Sales Executive" << endl; }
};
```

C++ Solution: Polymorphic Hierarchy (Flexible) Engineer + Manager + Director + Others

```
vector<Employee*> Employee::staffs;         // Collection of all staffs

int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
    Manager m1("Kamala"), m2("Rajib");
    SalesExecutive s1("Hari"), s2("Bishnu");
    Director d("Ranjana");

    vector<Employee*>::const_iterator it;   // Iterator over staffs

    for (it = Employee::staffs.begin();     // Iterate on staffs
            it < Employee::staffs.end();
            ++it)
        (*it)->ProcessSalary();             // Process respective salary
}
```

```
Rohit: Process Salary for Engineer
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Hari: Process Salary for Sales Executive
Bishnu: Process Salary for Sales Executive
Ranjana: Process Salary for Director
```

C++ Solution: Polymorphic Hierarchy (Flexible): Advantages and Disadvantages

- **Advantages**
  - Data is fully encapsulated
  - Flexible Polymorphic Hierarchy makes addition of any class possible on the hierarchy
  - Application code is independent of types in the system (`virtual` functions manage types through polymorphic dispatch)
  - Maximum Code reuse – code is short and simple
  - Collection of staff encapsulated with creation
  - `vector` and `iterator` increases efficiency and efficacy
- **Disadvantages**
  - None in particular
- **Recommendation**
  - Enjoy the solution

Module Summary

- Completed design for a staff salary problem using hierarchy and worked out extensible C++ solution
- Learnt about iterative refinement of solutions in the process