Module 27
Instructors: Abir Das and Jibesh Patra

Module 27: Programming in C++

Polymorphism: Part 2: Static and Dynamic Binding

Instructors: Abir Das and Jibesh Patra
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

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

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

by **Prof. Partha Pratim Das**

Module Objectives

- Understand Static and Dynamic Binding
- Understand Polymorphic Type

Module Outline

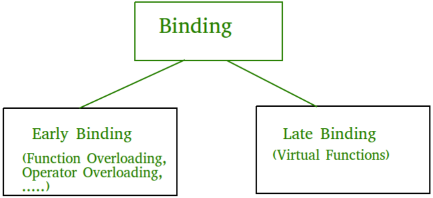
Type of an Object

- The *static type* of the object is the type declared for the object while writing the code
- Compiler *sees static type*
- The *dynamic type* of the object is determined by the type of the object to which it *refers at run-time*
- Compiler *does not see dynamic type*

```
class A { };
class B : public A { };

int main() {
    A *p;
    p = new B; // Static type of p is A*
               // Dynamic type of p is B*
}
```

Static and Dynamic Binding

- **Static binding** (early binding): When a function invocation binds to the function definition based on the static type of objects
  - This is done at *compile-time*
  - Normal *function calls*, *overloaded function calls*, and *overloaded operators* are examples of *static binding*
- **Dynamic binding** (late binding): When a function invocation binds to the function definition based on the dynamic type of objects
  - This is done at *run-time*
  - *Function pointers*, *Virtual functions* are examples of *late binding*



Comparison of Static and Dynamic Binding

Basis	Static Binding	Dynamic Binding
• **Event Occurrence** • **Information**	• Events occur at *compile time* – *Static Binding* • All information needed to call a function is known at *compile time*	• Events occur at *run time* – *Dynamic Binding* • All information needed to call a function is known only at *run time*
• **Advantage** • **Time** • **Actual Object**	• *Efficiency* • *Fast execution* • Actual object is *not used for binding*	• *Flexibility* • *Slow execution* • Actual object is *used for binding*
• **Alternate name**	• *Early Binding*	• *Late Binding*
• **Example**	• *Method Overloading* Normal function call, Overloaded function call, Overloaded operators	• *Method Overriding* Virtual functions

Static Binding

Inherited Method	Overridden Method

`#include<iostream>`
`using namespace std;`
`class B { public:`
`    void f() { }`
`};`
`class D : public B { public:`
`    void g() { } // new function`
`};`
`int main() { B b; D d;`
 	
`    b.f(); // B::f()`
`    d.f(); // B::f() ----- Inherited`
`    d.g(); // D::g() ----- Added`
`}`
	`#include<iostream>`
	`using namespace std;`
	`class B { public:`
	`    void f() { }`
	`};`
	`class D : public B { public:`
	`    void f() { }`
	`};`
	`int main() { B b; D d;`
	
	`    b.f(); // B::f()`
	`    d.f(); // D::f() ----- Overridden`
	`           // masks the base class function`
	`}`

• Object `d` of derived class inherits the base class function `f()` and has its own function `g()` • Function calls are resolved at compile time based on static type	• If a member function of a base class is redefined in a derived class with the same signature then it masks the base class method • The derived class method `f()` is linked to the object `d`. As `f()` is redefined in the derived class, the base class version cannot be called with the object of a derived class

Member Functions: Overrides and Overloads: RECAP (Module 22)

Inheritance

```
class B { public: // Base Class
    void f(int i);
    void g(int i);
};
class D: public B { public: // Derived Class
    // Inherits B::f(int)


    // Inherits B::g(int)

};
B b;
D d;

b.f(1); // Calls B::f(int)
b.g(2); // Calls B::g(int)

d.f(3); // Calls B::f(int)
d.g(4); // Calls B::g(int)
```

Override & Overload

```
class B { public: // Base Class
    void f(int);
    void g(int i);
};
class D: public B { public: // Derived Class
    // Inherits B::f(int)
    void f(int);      // Overrides B::f(int)
    void f(string&); // Overloads B::f(int)
    // Inherits B::g(int)
    void h(int i);   // Adds D::h(int)
};
B b;
D d;

b.f(1);      // Calls B::f(int)
b.g(2);      // Calls B::g(int)

d.f(3);      // Calls D::f(int)
d.g(4);      // Calls B::g(int)

d.f("red"); // Calls D::f(string&)
d.h(5);      // Calls D::h(int)
```

- D::f(int) overrides B::f(int)
- D::f(string&) overloads B::f(int)

using Construct – Avoid Method Hiding

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

class A { public:
    void f() { }
};

class B : public A { public:
     // To overload, rather than hide the base class function f(),
     // it is introduced into the scope of B with a using declaration
    using A::f;
    void f(int) { } // Overloads f()
};
int main() {
    B b; // function calls resolved at compile time

    b.f(3); // B::f(int)
    b.f();  // A::f()
}
```

- Object `b` of derived class linked to with inherited base class function `f()` and the overloaded version defined by the derived class `f(int)`, based on the input parameters – function calls resolved at compile time

Dynamic Binding

Non-Virtual Method

```
#include<iostream>
using namespace std;
class B { public:
    void f() { }
};
class D : public B { public:
    void f() { }
};
int main() {
    B b;
    D d;

    B *p;

    p = &b; p->f(); // B::f()
    p = &d; p->f(); // B::f()
}
```

- `p->f()` always binds to `B::f()`
- Binding is decided by the *type of pointer*
- **Static Binding**

Virtual Method

```
#include<iostream>
using namespace std;
class B { public:
    virtual void f() { }
};
class D : public B { public:
    virtual void f() { }
};
int main() {
    B b;
    D d;

    B *p;

    p = &b; p->f(); // B::f()
    p = &d; p->f(); // D::f()
}
```

- `p->f()` binds to `B::f()` for a `B` object, and to `D::f()` for a `D` object
- Binding is decided by the *type of object*
- **Dynamic Binding**

Static and Dynamic Binding

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

class B { public:
    void f() { cout << "B::f()" << endl; }
    virtual void g() { cout << "B::g()" << endl; }
};
class D: public B { public:
    void f() { cout << "D::f()" << endl; }
    virtual void g() { cout << "D::g()" << endl; }
};
```

```
int main() { B b; D d;

    B *pb = &b;
    B *pd = &d; // UPCAST

    B &rb = b;
    B &rd = d;  // UPCAST

    b.f(); // B::f()
    b.g(); // B::g()
    d.f(); // D::f()
    d.g(); // D::g()

    pb->f(); // B::f() -- Static Binding
    pb->g(); // B::g() -- Dynamic Binding
    pd->f(); // B::f() -- Static Binding
    pd->g(); // D::g() -- Dynamic Binding

    rb.f();  // B::f() -- Static Binding
    rb.g();  // B::g() -- Dynamic Binding
    rd.f();  // B::f() -- Static Binding
    rd.g();  // D::g() -- Dynamic Binding

    return 0;
}
```

Polymorphic Type: Virtual Functions

- *Dynamic binding* is possible only for pointer and reference data types and for member functions that are declared as `virtual` in the base class
- These are called Virtual Functions
- If a member function is declared as virtual, it can be overridden in the derived class
- If a member function is not virtual and it is re-defined in the derived class then the latter definition hides the former one
- Any class containing a virtual member function – by definition or by inheritance – is called a Polymorphic Type
- A hierarchy may be *polymorphic* or *non-polymorphic*
- A non-polymorphic hierarchy has little value

Polymorphism Rule

```
#include <iostream>
using namespace std;
class A { public:
    void f()         { cout << "A::f()" << endl; } // Non-Virtual
    virtual void g() { cout << "A::g()" << endl; } // Virtual
    void h()         { cout << "A::h()" << endl; } // Non-Virtual
};
class B : public A { public:
    void f()         { cout << "B::f()" << endl; } // Non-Virtual
    void g()         { cout << "B::g()" << endl; } // Virtual
    virtual void h() { cout << "B::h()" << endl; } // Virtual
};
class C : public B { public:
    void f()         { cout << "C::f()" << endl; } // Non-Virtual
    void g()         { cout << "C::g()" << endl; } // Virtual
    void h()         { cout << "C::h()" << endl; } // Virtual
};
```

```
int main() {
    B *q = new C; A *p = q;

    p->f();                q->f();
    p->g();                q->g();
    p->h();                q->h();
}
```

```
A::f()
C::g()
A::h()
B::f()
C::g()
C::h()
```

Module Summary

- Discussed Static and Dynamic Binding
- Polymorphic type introduced

Module 28: Programming in C++

Polymorphism: Part 3: Abstract Base Class

Instructors: Abir Das and Jibesh Patra
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

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

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

by **Prof. Partha Pratim Das**

Module Objectives

- Understand why destructor must be `virtual` in a class hierarchy
- Learn to work with class hierarchy

Module Outline

Virtual Destructor

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

class B { int data_; public:
    B(int d) :data_(d) { cout << "B()" << endl; }
    ~B() { cout << "~B()" << endl; }
     virtual void Print() { cout << data_; }
};
class D: public B { int *ptr_; public:
    D(int d1, int d2) :B(d1), ptr_(new int(d2)) { cout << "D()" << endl; }
    ~D() { cout << "~D()" << endl; delete ptr_; }
    void Print() { B::Print();  cout << " " << *ptr_; }
};
int main() {
    B *p = new B(2);
    B *q = new D(3, 5);

    p->Print(); cout << endl;
    q->Print(); cout << endl;

    delete p;
    delete q;
}
```

```
Output:
B()
B()
D()
2
3 5
~B()
~B()
```

Destructor of d (type D) not called!

Virtual Destructor

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

class B { int data_; public:
    B(int d) :data_(d) { cout << "B()" << endl; }
    virtual ~B() { cout << "~B()" << endl; }        // Destructor made virtual
    virtual void Print() { cout << data_; }
};
class D: public B { int *ptr_; public:
    D(int d1, int d2) :B(d1), ptr_(new int(d2)) { cout << "D()" << endl; }
    ~D() { cout << "~D()" << endl; delete ptr_; }
    void Print() { B::Print();  cout << " " << *ptr_; }
};
int main() {
    B *p = new B(2);
    B *q = new D(3, 5);

    p->Print(); cout << endl;
    q->Print(); cout << endl;

    delete p;
    delete q;
}
```

```
Output:
B()
B()
D()
2
3 5
~B()
~D()
~B()
```

Destructor of d (type D) is called!

Virtual Destructor: Slicing

- Slicing is where we assign an object of a derived class to an instance of a base class, thereby losing part of the information - some of it is sliced away

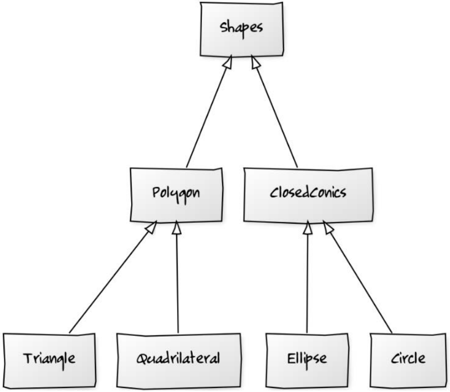
```
#include <iostream>
using namespace std;
class Base { protected: int i; public:
    Base(int a)      i = a;
    virtual void display() { cout << "I am Base class object, i = " << i << endl; }
};
class Derived : public Base { int j; public:
    Derived(int a, int b) : Base(a) { j = b; }
    virtual void display() { cout<< "I am Derived class object, i = " << i << ", j = " << j <<endl; }
};
// Global method, Base class object is passed by value
void somefunc (Base obj) { obj.display(); }
int main() { Base b(33); Derived d(45, 54);
    somefunc(b);
    somefunc(d);  // Object Slicing, the member j of d is sliced off
}
```

```
I am Base class object, i = 33
I am Base class object, i = 45
```

- If the destructor is not `virtual` in a polymorphic hierarchy, it leads to Slicing
- **Destructor must be declared `virtual` in the base class**

Pure Virtual Function

Hierarchy of Shapes



- We want to have a polymorphic `draw()` function for the hierarchy
- `draw()` will be overridden in every class based on the drawing algorithms
- What is the `draw()` function for the root `Shapes` class?

Pure Virtual Function

- For the polymorphic hierarchy of Shapes, we need draw() to be a virtual function
- draw() must be a member of Shapes class for polymorphic dispatch to work
- But we cannot define the body of draw() function for the root Shapes class as we do not have an algorithm to draw an arbitrary share. In fact, we cannot even have a representation for shapes in general!
- Pure Virtual Function solves the problem
- A Pure Virtual Function has a signature but no body!
- Example:

```
class Root { public:
    void f();               // Non-Virtual Function
    virtual void g();       // Virtual Function
    virtual void h() = 0;   // Pure Virtual Function
};
```

Abstract Base Class

Abstract Base Class

Abstract Base Class

- A class containing at least one Pure Virtual Function is called an Abstract Base Class
- Pure Virtual Functions may be inherited or defined in the class
- No instance can be created for an Abstract Base Class
- Naturally it may not have a constructor or a `virtual` destructor
- An Abstract Base Class, however, may have other `virtual` (non-pure) and non-`virtual` member functions as well as data members
- Data members in an Abstract Base Class should be protected. Of course, private and public data are also allowed
- Member functions in an Abstract Base Class should be public. Of course, private and protected methods are also allowed
- A Concrete Class must override and implement all Pure Virtual Functions so that it can be instantiated

Shape Hierarchy


```
#include <iostream>  // Abstract Base Class shown in red
using namespace std; // Concrete Class shown in green

class Shapes { public:                                   // Abstract Base Class
    virtual void draw() = 0; // Pure Virtual Function
};
class Polygon: public Shapes { public: void draw() { cout<< "Polygon: Draw by Triangulation" <<endl; } };
class ClosedConics: public Shapes { public:              // Abstract Base Class
    // draw() inherited - Pure Virtual
};
class Triangle: public Polygon { public: void draw() { cout << "Triangle: Draw by Lines" << endl; } };
class Quadrilateral: public Polygon { public:
    void draw() { cout << "Quadrilateral: Draw by Lines" << endl; }
};
class Circle: public ClosedConics { public:
    void draw() { cout << "Circle: Draw by Breshenham Algorithm" << endl; }
};
class Ellipse: public ClosedConics { public: void draw() { cout << "Ellipse: Draw by ..." << endl; } };
int main() {
    Shapes *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };

    for (int i = 0; i < sizeof(arr) / sizeof(Shapes *); ++i)
        arr[i]->draw();
    // ...
}
```

Shape Hierarchy

```
int main() {
    Shapes *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };

    for (int i = 0; i < sizeof(arr) / sizeof(Shapes *); ++i)
        arr[i]->draw();
    // ...
    return 0;
}
```

```
Triangle: Draw by Lines
Quadrilateral: Draw by Lines
Circle: Draw by Breshenham Algorithm
Ellipse: Draw by ...
```

- Instances for class `Shapes` and class `ClosedConics` cannot be created

Shape Hierarchy: A Pure Virtual Function may have a body!


```
#include <iostream>
using namespace std;
class Shapes { public:                                  // Abstract Base Class
    virtual void draw() = 0 // Pure Virtual Function
    { cout << "Shapes: Init Brush" << endl; }
};
class Polygon: public Shapes { public:                  // Concrete Class
    void draw() { Shapes::draw(); cout << "Polygon: Draw by Triangulation" << endl; }
};
class ClosedConics: public Shapes { public:             // Abstract Base Class
    // draw() inherited - Pure Virtual
};
class Triangle: public Polygon { public:                // Concrete Class
    void draw() { Shapes::draw(); cout << "Triangle: Draw by Lines" << endl; }
};
class Quadrilateral: public Polygon { public:           // Concrete Class
    void draw() { Shapes::draw(); cout << "Quadrilateral: Draw by Lines" << endl; }
};
class Circle: public ClosedConics { public:             // Concrete Class
    void draw() { Shapes::draw(); cout << "Circle: Draw by Breshenham Algorithm" << endl; }
};
class Ellipse: public ClosedConics { public:            // Concrete Class
    void draw() { Shapes::draw(); cout << "Ellipse: Draw by ..." << endl; }
};
```

Shape Hierarchy

```
int main() {
    Shapes *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };

    for (int i = 0; i < sizeof(arr) / sizeof(Shapes *); ++i)
        arr[i]->draw();
}
```

```
Shapes: Init Brush
Triangle: Draw by Lines
Shapes: Init Brush
Quadrilateral: Draw by Lines
Shapes: Init Brush
Circle: Draw by Breshenham Algorithm
Shapes: Init Brush
Ellipse: Draw by ...
```

- Instances for class Shapes and class ClosedConics cannot be created
- Some compilers do not allow to inline the function body for a pure virtual function

```
class Shapes { public: virtual void draw() = 0 { cout << "Shapes: Init Brush" << endl; } };
```

  Outline the function body:

```
class Shapes { public: virtual void draw() = 0; };
void Shapes::draw() { cout << "Shapes: Init Brush" << endl; }
```

Module Summary

- Discussed why destructors must be `virtual` in a polymorphic hierarchy
- Introduced Pure Virtual Functions
- Introduced Abstract Base Class

Module 29: Programming in C++

Polymorphism: Part 4: Staff Salary Processing using C

Instructors: Abir Das and Jibesh Patra
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

{abir, jibesh}@cse.iitkgp.ac.in

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

by **Prof. Partha Pratim Das**

Module Objectives

- Understand design with ISA related concepts
- Understand the problems with C design

Module Outline

Binding: Exercise 1

```
// Class Definitions
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int h(A *) { }
};
class B: public A { public:
    void f(int) { }
    virtual int h(B *) { }
};
class C: public B { public:
    void g(double) { }
    int h(B *) { }
};
```

```
// Application Codes
A a;
B b;
C c;

A *pA;
B *pB;
```

	Initialization		
Invocation	pA = &a;	pA = &b;	pA = &c;
pA->f(2);			
pA->g(3.2);			
pA->h(&a);			
pA->h(&b);			

Binding: Exercise 1: Solution

```
// Class Definitions
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int h(A *) { }
};
class B: public A { public:
    void f(int) { }
    virtual int h(B *) { }
};
class C: public B { public:
    void g(double) { }
    int h(B *) { }
};
```

```
// Application Codes
A a;
B b;
C c;

A *pA;
B *pB;
```

	Initialization		
Invocation	pA = &a;	pA = &b;	pA = &c;
pA->f(2);	A::f	B::f	B::f
pA->g(3.2);	A::g	A::g	C::g
pA->h(&a);	A::h	A::h	A::h
pA->h(&b);	A::h	A::h	A::h

Binding: Exercise 2


```
// Class Definitions
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int h(A *) { }
};
class B: public A { public:
    void f(int) { }
    virtual int h(B *) { }
};
class C: public B { public:
    void g(double) { }
    int h(B *) { }
};
```

```
// Application Codes
A a;
B b;
C c;

A *pA;
B *pB;
```

Invocation	Initialization pB = &a;	pB = &b;	pB = &c;
pB->f(2);			
pB->g(3.2);			
pB->h(&a);			
pB->h(&b);			

Binding: Exercise 2: Solution


```
// Class Definitions
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int h(A *) { }
};
class B: public A { public:
    void f(int) { }
    virtual int h(B *) { }
};
class C: public B { public:
    void g(double) { }
    int h(B *) { }
};
```

```
// Application Codes
A a;
B b;
C c;

A *pA;
B *pB;
```

	Initialization		
Invocation	pB = &a;	pB = &b;	pB = &c;
pB->f(2);	Error Downcast (A *) to (B *)	B::f	B::f
pB->g(3.2);		A::g	C::g
pB->h(&a);		No conversion (A *) to (B *)	
pB->h(&b);		B::h	C::h

Staff Salary Processing: Problem Statement

- An organization needs to develop a salary processing application for its staff
- At present it has an engineering division only where Engineers and Managers work. Every Engineer reports to some Manager. Every Manager can also work like an Engineer
- The logic for processing salary for Engineers and Managers are different as they have different salary heads
- In future, it may add Directors to the team. Then every Manager will report to some Director. Every Director could also work like a Manager
- The logic for processing salary for Directors will also be distinct
- Further, in future it may open other divisions, like Sales division, and expand the workforce
- **Make a suitable extensible design**

C Solution: Function Switch: Engineer + Manager

- How to represent Engineers and Managers?
  - Collection of `struct`s
- How to initialize objects?
  - Initialization functions
- How to have a collection of mixed objects?
  - Array of `union`
- How to model variations in salary processing algorithms?
  - `struct`-specific functions
- How to invoke the correct algorithm for a correct employee type?
  - Function Switch
  - Function Pointers

C Solution: Function Switch: Engineer + Manager


```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef enum E_TYPE { Er, Mgr } E_TYPE; // Tag for type of staff

typedef struct Engineer { char *name_; } Engineer;
Engineer *InitEngineer(const char *name) {
    Engineer *e = (Engineer *)malloc(sizeof(Engineer));
    e->name_ = strdup(name); return e;
}
void ProcessSalaryEngineer(Engineer *e) { printf("%s: Process Salary for Engineer\n", e->name_); }

typedef struct Manager { char *name_; Engineer *reports_[10]; } Manager;
Manager *InitManager(const char *name) {
    Manager *m = (Manager *)malloc(sizeof(Manager));
    m->name_ = strdup(name); return m;
}
void ProcessSalaryManager(Manager *m) { printf("%s: Process Salary for Manager\n", m->name_); }

typedef struct Staff { // Aggregation of staffs
    E_TYPE type_;
    union { Engineer *pE; Manager *pM; };
} Staff;
```

C Solution: Function Switch: Engineer + Manager

```
int main() {
    Staff allStaff[10];
    allStaff[0].type_ = Er;  allStaff[0].pE = InitEngineer("Rohit");
    allStaff[1].type_ = Mgr; allStaff[1].pM = InitManager("Kamala");
    allStaff[2].type_ = Mgr; allStaff[2].pM = InitManager("Rajib");
    allStaff[3].type_ = Er;  allStaff[3].pE = InitEngineer("Kavita");
    allStaff[4].type_ = Er;  allStaff[4].pE = InitEngineer("Shambhu");

    for (int i = 0; i < 5; ++i) {
        E_TYPE t = allStaff[i].type_;
        if (t == Er)
            ProcessSalaryEngineer(allStaff[i].pE);
        else if (t == Mgr)
            ProcessSalaryManager(allStaff[i].pM);
         else
            printf("Invalid Staff Type\n");
    }
}
```

```
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: Function Switch: Engineer + Manager + Director

- How to represent Engineers, Managers, and Directors?
  - Collection of `struct`s
- How to initialize objects?
  - Initialization functions
- How to have a collection of mixed objects?
  - Array of `union`
- How to model variations in salary processing algorithms?
  - `struct`-specific functions
- How to invoke the correct algorithm for a correct employee type?
  - Function switch
  - Function pointers

C Solution: Function Switch: Engineer + Manager + Director

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef enum E_TYPE { Er, Mgr, Dir } E_TYPE;

typedef struct Engineer { char *name_; } Engineer;
Engineer *InitEngineer(const char *name) { Engineer *e = (Engineer *)malloc(sizeof(Engineer));
    e->name_ = strdup(name); return e;
}
void ProcessSalaryEngineer(Engineer *e) { printf("%s: Process Salary for Engineer\n", e->name_); }

typedef struct Manager { char *name_; Engineer *reports_[10]; } Manager;
Manager *InitManager(const char *name) { Manager *m = (Manager *)malloc(sizeof(Manager));
    m->name_ = strdup(name); return m;
}
void ProcessSalaryManager(Manager *m) { printf("%s: Process Salary for Manager\n", m->name_); }

typedef struct Director { char *name_; Manager *reports_[10]; } Director;
Director *InitDirector(const char *name) { Director *d = (Director *)malloc(sizeof(Director));
    d->name_ = strdup(name); return d;
}
void ProcessSalaryDirector(Director *d) { printf("%s: Process Salary for Director\n", d->name_); }

typedef struct Staff { E_TYPE type_; union { Engineer *pE; Manager *pM; Director *pD; };
} Staff;
```

C Solution: Function Switch: Engineer + Manager + Director


```
int main() { Staff allStaff[10];
    allStaff[0].type_ = Er;  allStaff[0].pE = InitEngineer("Rohit");
    allStaff[1].type_ = Mgr; allStaff[1].pM = InitManager("Kamala");
    allStaff[2].type_ = Mgr; allStaff[2].pM = InitManager("Rajib");
    allStaff[3].type_ = Er;  allStaff[3].pE = InitEngineer("Kavita");
    allStaff[4].type_ = Er;  allStaff[4].pE = InitEngineer("Shambhu");
    allStaff[5].type_ = Dir; allStaff[5].pD = InitDirector("Ranjana");

    for (int i = 0; i < 6; ++i) { E_TYPE t = allStaff[i].type_;
        if (t == Er)
            ProcessSalaryEngineer(allStaff[i].pE);
        else if (t == Mgr)
            ProcessSalaryManager(allStaff[i].pM);
        else if (t == Dir)
            ProcessSalaryDirector(allStaff[i].pD);
        else
            printf("Invalid Staff Type\n");
    }
}
```

```
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: Function Switch: Engineer + Manager + Director

Instead of if-else chain, we can use switch to explicitly switch on the type of employee

```
int main() { Staff allStaff[10];
    allStaff[0].type_ = Er;  allStaff[0].pE = InitEngineer("Rohit");
    allStaff[1].type_ = Mgr; allStaff[1].pM = InitManager("Kamala");
    allStaff[2].type_ = Mgr; allStaff[2].pM = InitManager("Rajib");
    allStaff[3].type_ = Er;  allStaff[3].pE = InitEngineer("Kavita");
    allStaff[4].type_ = Er;  allStaff[4].pE = InitEngineer("Shambhu");
    allStaff[5].type_ = Dir; allStaff[5].pD = InitDirector("Ranjana");

    for (int i = 0; i < 6; ++i) { E_TYPE t = allStaff[i].type_;
        switch (t) {
            case Er:  ProcessSalaryEngineer(allStaff[i].pE); break;
            case Mgr: ProcessSalaryManager(allStaff[i].pM); break;
            case Dir: ProcessSalaryDirector(allStaff[i].pD); break;
            default:  printf("Invalid Staff Type\n"); break;
        }
    }
}
```

```
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: Advantages and Disadvantages

- **Advantages**
  - Solution exists!
  - Code is well structured – has patterns
- **Disadvantages**
  - Employee data has scope for better organization
    - No encapsulation for data
    - Duplication of fields across types of employees – possible to mix up types for them (say, `char *` and `string`)
    - Employee objects are created and initialized dynamically through `Init...` functions. How to release the memory?
  - 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`
      - Add a new pointer field in `struct Staff` for the new type
      - Add a new case (`if-else` or `case`) based on the new type
    - Error prone – developer has to decide to call the right processing function for every type (`ProcessSalaryManager` for `Mgr` etc.)
- **Recommendation**
  - Use classes for encapsulation on a hierarchy

Module Summary

- Practiced exercise with binding – various mixed cases
- Started designing for a staff salary problem and worked out C solutions

Module 30: Programming in C++

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

Instructors: Abir Das and Jibesh Patra

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

{abir, jibesh}@cse.iitkgp.ac.in

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

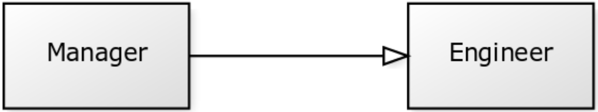
by **Prof. Partha Pratim Das**

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

```
#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


```
#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

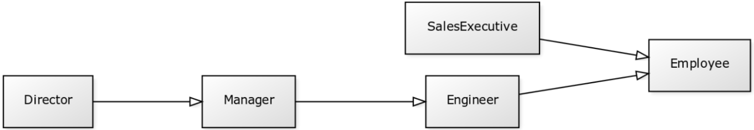

```
int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
    Manager m1("Kamala"), m2("Rajib");
    Director d("Ranjana");
    Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3, &d };

    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

Module 31: Programming in C++

Virtual Function Table

Instructors: Abir Das and Jibesh Patra
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

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

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

by **Prof. Partha Pratim Das**

Weekly Recap

- Understood type casting – implicit as well as explicit – for built-in types, unrelated types, and classes on a hierarchy
- Understood the notions of upcast and downcast
- Understood Static and Dynamic Binding for Polymorphic type
- Understood `virtual` destructors, Pure Virtual Functions, and Abstract Base Class
- Designed the solution for a staff salary processing problem using iterative refinement – starting with a simple C solution and repeatedly refining finally to an easy, efficient, and extensible C++ solution based on flexible polymorphic hierarchy

Module Objectives

- Introduce a new C solution with function pointers
- Understand Virtual Function Table for dynamic binding (polymorphic dispatch)

Module Outline

Staff Salary Processing: New C Solution

Staff Salary Processing: Problem Statement: RECAP (Module 29)

- An organization needs to develop a salary processing application for its staff
- At present it has an engineering division only where Engineers and Managers work. Every Engineer reports to some Manager. Every Manager can also work like an Engineer
- The logic for processing salary for Engineers and Managers are different as they have different salary heads
- In future, it may add Directors to the team. Then every Manager will report to some Director. Every Director could also work like a Manager
- The logic for processing salary for Directors will also be distinct
- Further, in future it may open other divisions, like Sales division, and expand the workforce
- **Make a suitable extensible design**

C Solution: Function Pointers
Engineer + Manager + Director: RECAP (Module 29)

- How to represent Engineers, Managers, and Directors?
  - Collection of `struct`s
- How to initialize objects?
  - Initialization functions
- How to have a collection of mixed objects?
  - Array of `union`
- How to model variations in salary processing algorithms?
  - `struct`-specific functions
- How to invoke the correct algorithm for a correct employee type?
  - Function switch
  - **Function pointers**

C Solution: Function Pointers: Engineer + Manager + Director

- In Module 29, we have developed a flat C Solution using *function switch*
- In Module 30, we refined the C Solution to develop two types of C++ Solution using
  - Non-polymorphic hierarchy - employing *function switch*
  - Polymorphic hierarchy - eomploying *virtual function*
- In Module 29, we had mentioned that in the flat C Solution it is not easy to use function pointers as the processing functions `void ProcessSalaryEngineer(Engineer *)`, `void ProcessSalaryManager(Manager *)`, and `void ProcessSalaryDirector(Director *)` all have different types of arguments and therefore a common function pointer type cannot be defined
- We can work around this by:
  - Passing the staff object as `void *`, instead of `Engineer *`, `Manager *`, or `Director *`
  - Cast it to respective object type in the respective function. That is, cast to `Engineer *` in `ProcessSalaryEngineer(Engineer *)` and so on
  - We can then use a function pointer type `void (*)(void *)`
- We illustrate in the Solution

C Solution: Function Pointers: Engineer + Manager + Director


```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef enum E_TYPE { Er, Mgr, Dir } E_TYPE; // Staff tag type
typedef void (*psFuncPtr)(void *);          // Processing func. ptr. type, passing the object by void *
typedef struct Engineer { char *name_; } Engineer; // Engineer Type
Engineer *InitEngineer(const char *name) { Engineer *e = (Engineer *)malloc(sizeof(Engineer));
    e->name_ = strdup(name); return e;
}
void ProcessSalaryEngineer(void *v) { Engineer *e = (Engineer *)v; // Cast explicitly to the staff object
    printf("%s: Process Salary for Engineer\n", e->name_);
}
typedef struct Manager { char *name_; Engineer *reports_[10]; } Manager; // Manager Type
Manager *InitManager(const char *name) { Manager *m = (Manager *)malloc(sizeof(Manager));
    m->name_ = strdup(name); return m;
}
void ProcessSalaryManager(void *v) { Manager *m = (Manager *)v; // Cast explicitly to the staff object
    printf("%s: Process Salary for Manager\n", m->name_);
}
typedef struct Director { char *name_; Manager *reports_[10]; } Director; // Director Type
Director *InitDirector(const char *name) { Director *d = (Director *)malloc(sizeof(Director));
    d->name_ = strdup(name); return d;
}
void ProcessSalaryDirector(void *v) { Director *d = (Director *)v; // Cast explicitly to the staff object
    printf("%s: Process Salary for Director\n", d->name_);
}
```

C Solution: Function Pointers: Engineer + Manager + Director


```
typedef struct Staff {
    E_TYPE type_; // Staff tag type
    void *p;      // Pointer to staff object
} Staff;          // Staff object wrapper
int main() {
    // Array of function pointers
    psFuncPtr psArray[] = { ProcessSalaryEngineer,  ProcessSalaryManager, ProcessSalaryDirector };

    // Array of staffs
    Staff staff[] = { { Er, InitEngineer("Rohit") }, { Mgr, InitEngineer("Kamala") },
                      { Mgr, InitEngineer("Rajib") }, { Er, InitEngineer("Kavita") },
                      { Er, InitEngineer("Shambhu") }, { Dir, InitEngineer("Ranjana") } };

    for (int i = 0; i < sizeof(staff) / sizeof(Staff); ++i)
        psArray[staff[i].type_] // Pick the right processing function for the tag - staff type
            (staff[i].p);       // Pass the pointer to the object - implicitly cast to void*
}
```

```
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: Advantages and Disadvantages: RECAP (Module 26) Annotated for Function Pointers

- **Advantages**
  - Solution exists!
  - Code is well structured – has patterns
- **Disadvantages**
  - Employee data has scope for better organization
    - No encapsulation for data
    - Duplication of fields across types of employees – possible to mix up types for them (say, `char *` and `string`)
    - Employee objects are created and initialized dynamically through `Init...` functions. How to release the memory?
  - 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`
      - Add a new pointer field in `struct Staff` for the new type
      - Add a new case (`if-else` or `case`) based on the new type: **Removed using function pointer**
    - Error prone – developer has to decide to call the right processing function for every type (`ProcessSalaryManager` for `Mgr` etc.): **Removed using function pointer**
  - Unable to use Function Pointers as each processing function takes a parameter of different type - no common signature for dispatch
- **Recommendation**
  - Use classes for encapsulation on a hierarchy

Staff Salary Processing: C++ Solution

C++ Solution: Polymorphic Hierarchy: RECAP Engineer + Manager + Director: (Module 30)

- 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: RECAP Engineer + Manager + Director: (Module 30)

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

class Engineer {
protected:
    string name_;
public:
    Engineer(const string& name) : name_(name) { }
    virtual ~Engineer() { }
    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: RECAP Engineer + Manager + Director: (Module 30)


```
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 and C++ Solutions: A Comparison

C and C++ Solutions: A Comparison

C Solution	C++ Solution

• How to represent Engineers, Managers, and Directors?
  ◦ structs
	• How to represent Engineers, Managers, and Directors?
	  ◦ Polymorphic hierarchy

• How to initialize objects?
  ◦ Initialization functions
	• How to initialize objects?
	  ◦ Ctor / Dtor

• How to have a collection of mixed objects?
  ◦ array of union wrappers
	• How to have a collection of mixed objects?
	  ◦ array of base class pointers

• How to model variations in salary processing algorithms?
  ◦ functions for structs
	• How to model variations in salary processing algorithms?
	  ◦ class member functions

• How to invoke the correct algorithm for a correct employee type?
  ◦ Function pointers
	• How to invoke the correct algorithm for a correct employee type?
	  ◦ Virtual Functions

C and C++ Solutions: A Comparison

C Solution (Function Pointer)

```
typedef enum E_TYPE { Er, Mgr, Dir } E_TYPE;
typedef void (*psFuncPtr)(void *);
typedef struct { E_TYPE type_; void *p; } Staff;
typedef struct { char *name_; } Engineer;
Engineer *InitEngineer(const char *name);
void ProcessSalaryEngineer(void *v);
typedef struct { char *name_; } Manager;
Manager *InitManager(const char *name);
void ProcessSalaryManager(void *v);
typedef struct { char *name_; } Director;
Director *InitDirector(const char *name);
void ProcessSalaryDirector(void *v);
int main() { psFuncPtr psArray[] = {
    ProcessSalaryEngineer,    // Function
    ProcessSalaryManager,     // pointer
    ProcessSalaryDirector };  // array
    Staff staff[] = {
    { Er, InitEngineer("Rohit") },
    { Mgr, InitEngineer("Kamala") },
    { Dir, InitEngineer("Ranjana") } };
    for (int i = 0; i <
        sizeof(staff)/sizeof(Staff); ++i)
        psArray[staff[i].type_](staff[i].p);
}
```

C++ Solution (Virtual Function)

```
class Engineer { protected: string name_;
public: Engineer(const string& name);
     virtual void ProcessSalary(); };
     virtual ~Engineer(); };
class Manager : public Engineer {
public: Manager(const string& name);
    void ProcessSalary(); };
class Director : public Manager {
public: Director(const string& name);
    void ProcessSalary(); };
int main() {
    // Function pointer array is subsumed in
    // virtual function tables of classes

    Engineer e1("Rohit");
    Manager m1("Kamala");
    Director d("Ranjana");
    Engineer *staff[] = { &e1, &m1, &d };
    for(int i = 0; i <
        sizeof(staff)/sizeof(Engineer*); ++i)
        staff[i]->ProcessSalary();
}
```

Virtual Function Pointer Table

How do virtual functions work?

- The C Solution with function pointers gives us the lead to implement virtual functions. Here
  - We have used an array of function pointers (`psFuncPtr psArray[]`) to keep the processing functions (`void ProcessSalaryEngineer(Engineer *)`, `void ProcessSalaryManager(Manager *)`, and `void ProcessSalaryDirector(Director *)`) indexed by the type tag (`enum E_TYPE { Er, Mgr, Dir }`)
  - In C++, every class is a separate type - so the tag can be removed if we bind this table (**Virtual Function Table** or VFT) with the class
  - Every class can have a VFT with its appropriate processing function pointer put there
  - By override, all these functions can have the same signature (`void ProcessSalary()`) and can be called through the same expression (`(Engineer *)->ProcessSalary()`)
- We now illustrate Virtual Function Table through simple examples to show how does it work for inherited, overridden and overloaded member functions

Virtual Function Pointer Table

Base Class	Derived Class

Base Class

```
class B {
    int i;
public:
    B(int i_): i(i_) { }
        void f(int); // B::f(B*const, int)
virtual void g(int); // B::g(B*const, int)
};

B b(100);
B *p = &b;
```

`b` **Object Layout**

	Object		*VFT*
vft	→	0	B::g(B*const, int)
B::i	100		

Source Expression	Compiled Expression
b.f(15);	B::f(&b, 15);
p->f(25);	B::f(p, 25);
b.g(35);	B::g(&b, 35);
p->g(45);	p->vft[0](p, 45);

Derived Class

```
class D: public B {
    int j;
public:
    D(int i_, int j_): B(i_), j(j_) { }
        void f(int); // D::f(D*const, int)
        void g(int); // D::g(D*const, int)
};

D d(200, 500);
B *p = &d;
```

`d` **Object Layout**

	Object		*VFT*
vft	→	0	D::g(D*const, int)
B::i	200		
D::j	500		

Source Expression	Compiled Expression
d.f(15);	D::f(&d, 15);
p->f(25);	B::f(p, 25);
d.g(35);	D::g(&d, 35);
p->g(45);	p->vft[0](p, 45);

Virtual Function Pointer Table

- Whenever a class defines a `virtual` function a hidden member variable is added to the class which points to an array of pointers to (`virtual`) functions called the **Virtual Function Table** (VFT)
- VFT pointers are used at run-time to invoke the appropriate function implementations, because at compile time it may not yet be known if the base function is to be called or a derived one implemented by a class that inherits from the base class
- VFT is class-specific – all instances of the class has the same VFT
- VFT carries the **Run-Time Type Information** (RTTI) of objects

Virtual Function Pointer Table

```
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int h(A *) { }
};
class B: public A { public:
    void f(int) { }
    virtual int h(B *) { }
};
class C: public B { public:
    void g(double) { }
    int h(B *) { }
};
A a; B b; C c;
A *pA; B *pB;
```

Source Expression	Compiled Expression
pA->f(2);	pA->vft[0](pA, 2);
pA->g(3.2);	pA->vft[1](pA, 3.2);
pA->h(&a);	A::h(pA, &a);
pA->h(&b);	A::h(pA, &b);
pB->f(2);	pB->vft[0](pB, 2);
pB->g(3.2);	pB->vft[1](pB, 3.2);
pB->h(&a);	pB->vft[2](pB, &a);
pB->h(&b);	pB->vft[2](pB, &b);

a Object Layout

Object: vft →

	VFT	
0	A::f(A*const, int)	Defined
1	A::g(A*const, double)	Defined

b Object Layout

Object: vft →

	VFT	
0	B::f(B*const, int)	Overridden
1	A::g(A*const, double)	Inherited
2	B::h(B*const, B*)	Overloaded

c Object Layout

Object: vft →

	VFT	
0	B::f(B*const, int)	Inherited
1	C::g(C*const, double)	Overridden
2	C::h(C*const, B*)	Overridden

Module Summary

- Leveraging an innovative solution to the Salary Processing Application in C using function pointers, we compare C and C++ solutions to the problem
- The new C solution with function pointers is used to explain the mechanism for dynamic binding (polymorphic dispatch) based on `virtual` function tables