Object Oriented Programming with C++

12. Dynamic binding

By: Prof. Pandav Patel

Second Semester, 2020-21
Computer Engineering Department
Dharmsinh Desai University

```
class A {
    int num;
public:
    A(int n) {
        num = n;
    }
    void print() {
        cout << num << endl;
    }
};
```

```
int main() {
    A a(5);
    int i;
    cout << "Size of integer : " << sizeof(int) << endl;
    // Methods does not contribute to the size of the object
    cout << "Size of A object: " << sizeof(a) << endl;

    A *ap = &a;
    a.print();
    // Methods can also be called using pointer to object
    ap->print();
    (*ap).print();
    return 0;
}
```

Size of integer : 4
Size of A object: 4
5
5
5

```
class A {
    int num;
    float flt;
public:
    A(int n = 0, float f = 1.1) {
        num = n;
        flt = f;
    }
    void print() {
        cout << num << " " << flt << endl;
    }
    void set_values(int n, float f) {
        num = n;
        flt = f;
    }
};
```

```
int main() {
    A a[5];
    cout << "Size of one A object : " << sizeof(a[0]) << endl;
    cout << "Size of Array of 5 A objects: " << sizeof(a) << endl;

    A *ap = a;
    a[0].print();
    // Methods can also be called using pointer to object
    ap->print();
    (*ap).print();
    a[2].set_values(2, 3.3);
    (ap + 2)->print();
    cout << ap << endl;
    ap += 2;
    cout << ap << endl;
    (ap)->print();

    return 0;
}
```

Size of one A object : 8
Size of Array of 5 A objects: 40
0 1.1
0 1.1
0 1.1
2 3.3
0x7ffc5127a690
0x7ffc5127a6a0
2 3.3

```
class A {
    int num;
    float flt;
public:
    A(int n = 0, float f = 1.1) {
        num = n;
        flt = f;
    }
    void print() {
        cout << "A:" << num << " " << flt << endl;
    }
    void set_values(int n, float f) {
        num = n;
        flt = f;
    }
};

class B {
    int num;
    float flt;
public:
    B(int n = 0, float f = 1.1) {
        num = n;
        flt = f;
    }
    void print() {
        cout << "B:" << num << " " << flt << endl;
    }
    void set_values(int n, float f) {
        num = n;
        flt = f;
    }
};
```

```
int main() {
    A a(1, 2.2), *a_p = &a;
    B b(3, 4.4), *b_p = &b;
    cout << "Size of one A object : " << sizeof(a) << endl;
    cout << "Size of one B objects: " << sizeof(b) << endl;

    //a = b; // error
    //b = a; // error
    //a_p = b_p; // error
    //b_p = a_p; // error
    cout << a_p << endl << b_p << endl;
    a_p->print();
    b_p->print();
    // Works, but must be avoided
    // Coz may result in unexpected result
    a_p = (A *)b_p;
    cout << "After casting\n";
    cout << a_p << endl << b_p << endl;
    a_p->print();
    b_p->print();
    return 0;
}
```

```
Size of one A object : 8
Size of one B objects: 8
0x7fffb3d7e428
0x7fffb3d7e430
A:1 2.2
B:3 4.4
After casting
0x7fffb3d7e430
0x7fffb3d7e430
A:3 4.4
B:3 4.4
```

```
class A {
    int num;
    float flt;
public:
    A(int n = 0, float f = 1.1) {
        num = n;
        flt = f;
    }
    void print() {
        cout << "A:" << num << " " << flt << endl;
    }
    void set_values(int n, float f) {
        num = n;
        flt = f;
    }
};

class B {
    float flt;
    int num;
public:
    B(int n = 0, float f = 1.1) {
        num = n;
        flt = f;
    }
    void print() {
        cout << "B:" << num << " " << flt << endl;
    }
    void set_values(int n, float f) {
        num = n;
        flt = f;
    }
};
```

```
int main() {
    A a(1, 2.2), *a_p = &a;
    B b(3, 4.4), *b_p = &b;
    cout << "Size of one A object : " << sizeof(a) << endl;
    cout << "Size of one B objects: " << sizeof(b) << endl;

    //a = b; // error
    //b = a; // error
    //a_p = b_p; // error
    //b_p = a_p; // error
    cout << a_p << endl << b_p << endl;
    a_p->print();
    b_p->print();
    // Works, but must be avoided
    // Coz may result in unexpected result
    a_p = (A *)b_p;
    cout << "After casting\n";
    cout << a_p << endl << b_p << endl;
    a_p->print();
    b_p->print();
    return 0;
}
```

```
Size of one A object : 8
Size of one B objects: 8
0x7ffe425880b8
0x7ffe425880c0
A:1 2.2
B:3 4.4
After casting
0x7ffe425880c0
0x7ffe425880c0
A:1082969293 4.2039e-45
B:3 4.4
```

```
class A {
public:
    int num1;
    float flt1;
    A(int n = 0, float f = 1.1) {
        num1 = n;
        flt1 = f;
    }
    void print() {
        cout << "A:" << num1 << " " << flt1 << endl;
    }
    void set_values(int n, float f) {
        num1 = n;
        flt1 = f;
    }
};

class B: public A {
    int num2;
    float flt2;
public:
    B(int n = 0, float f = 1.1) {
        num2 = n;
        flt2 = f;
    }
    void print() {
        cout << "B:" << num1 << " " << flt1 << " ";
        cout << num2 << " " << flt2 << endl;
    }
    void set_values(int n, float f) {
        num2 = n;
        flt2 = f;
    }
};
```

```
int main() {
    A a(1, 2.2), *a_p = &a;
    B b(3, 4.4), *b_p = &b;
    cout << "Size of one A object : " << sizeof(a) << endl;
    cout << "Size of one B objects: " << sizeof(b) << endl;

    a_p->print();
    b_p->print();

    a = b; // Object slicing
    //b = a; // error
    cout << "After a = b;\n";
    a_p->print();
    b_p->print();

    b.A::set_values(6, 7.7);
    cout << "After b.A::set_values(6, 7.7);\n";
    a_p->print();
    b_p->print();

    a_p = b_p; //upcast
    //b_p = a_p; // error - downcast
    //b_p = (B *)a_p; //would work but must be avoided
    cout << "After a_p = b_p;\n";
    a_p->print();
    b_p->print();

    cout << a_p->num1 << " " << a_p->flt1 << endl;
    // Error
    //cout << a_p->num2 << " " << a_p->flt2 << endl;

    return 0;
}
```

```
Size of one A object : 8
Size of one B objects: 16
A:1 2.2
B:0 1.1 3 4.4
After a = b;
A:0 1.1
B:0 1.1 3 4.4
After b.A::set_values(6, 7.7);
A:0 1.1
B:6 7.7 3 4.4
After a_p = b_p;
A:6 7.7
B:6 7.7 3 4.4
6 7.7
```

```cpp
// q1.cpp
#include<iostream>

using std::cout;

class B {
    int i = 2;
public:
    void print() {
        cout << "i = " << i << "\n";
    }
};

class D: public B {
    int j = 3;
public:
    void print() {
        cout << "j = " << j << "\n";
    }
};

int main() {
    B b;
    D d;

    b.print();
    d.print();

    return 0;
}
```

b.print() will call print method of B class
d.print() will call print method of D class

i = 2
j = 3

```cpp
// q2.cpp
#include<iostream>

using std::cout;

class B {
    int i = 2;
public:
    void print() {
        cout << "i = " << i << "\n";
    }
};

class D: public B {
    int i = 3;
public:
    void print() {
        cout << "i = " << i << "\n";
    }
};

int main() {
    B b;
    D d;

    b.print();
    d.print();

    return 0;
}
```

b.print() will call print method of B class
d.print() will call print method of D class

Value of member i for D class is 3
Value of member i for B class is 2

From methods of D class member i of base class can be accessed using B::i

i = 2
i = 3

```cpp
// q3.cpp
#include<iostream>

using std::cout;

class B {
public:
    int i = 2;
    void print() {
        cout << "i = " << i << "\n";
    }
};

class D: public B {
    int i = 3;
public:
    void print() {
        cout << "B::i = " << B::i << "\n";
    }
};

int main() {
    B b;
    D d;

    b.print();
    d.print();

    cout << "from main: d.B::i = " << d.B::i << "\n";
    // error: ‘int D::i’ is private within this context
    // cout << "from main: d.i = " << d.i << "\n";

    return 0;
}
```

i = 2
B::i = 2
from main: d.B::i = 2

b.print() will call print method of B class
d.print() will call print method of D class

Value of member i for D class is 3
Value of member i for B class is 2

From methods of D class member i of
base class can be accessed using B::i

```cpp
// q4.cpp
#include<iostream>

using std::cout;

class B {
public:
    int i = 2;
    void print() {
        cout << "i = " << i << "\n";
    }
};

class D: public B {
    int i = 3;
public:
    void print() {
        cout << "B::i = " << B::i << "\n";
    }
};

int main() {
    B b;
    D d;

    b.print();
    d.print();
    d.B::print();

    return 0;
}
```

Method print of D class will be called when its object invokes print method.

If we want to invoke method of base class (B in this case) using object of derived class (b in this case), then it can be done by providing full-qualified name after dot operator.
e.g. d.B::print() will invoke print method of B class.

```
i = 2
B::i = 2
i = 2
```

```cpp
// q4.5.cpp
#include<iostream>

using std::cout;

class B {
public:
    int i = 2;
    void print() {
        cout << "i = " << i << "\n";
    }
};

class D: public B {
};

int main() {
    B b;
    D d;

    b.print();
    d.print();
    d.B::print();

    return 0;
}
```

If derived class is not declaring data member of member function (method) with the same name as in base class then they can still be accessed without full-qualified name.

For example here class D does not have data member i and print method of its own, but it still has inherited member i and method print. And hence inherited members can be accessed from methods of D class without full-qualiied name and they can also be invoked using object of D class

```
i = 2
i = 2
i = 2
```

```
// q5.cpp
#include<iostream>

using std::cout;

class B {
public:
    int i = 2;
    void print() {
        cout << "i = " << i << "\n";
    }
};

class D: public B {
    int i = 3;
public:
    void print(int x) {
        cout << "B::i = " << B::i << "\nx = " << x << "\n";
    }
};

int main() {
    B b;
    D d;

    // error: no matching function for call to ‘B::print(int)’
    // b.print(7);
    d.print(5);

    return 0;
}
```

B::i = 2
x = 5

Here base class B only has one print() method. So object of B can not invoke print(int) which is not part of class B.

```
// q6.cpp
#include<iostream>

using std::cout;

class B {
public:
    int i = 2;
    void print() {
        cout << "i = " << i << "\n";
    }
};

class D: public B {
    int i = 3;
public:
    void print(int x) {
        cout << "B::i = " << B::i << " x = " << x << "\n";
    }
};

int main() {
    B b;
    D d;

    b.print();
    // error: no matching function for call to ‘D::print()’
    // d.print();

    return 0;
}
```

i = 2

Here derived class D is having print method with one argument. So class D has two print methods B::print() and D::print(int).

From methods of class D, we can only invoke print(int) defined D without full qualification. And same is true for object of class D. Object of class D can invoke only print(int) (which is defined in class D) without full-qualified name.

If we want to invoke B::print() from methods of class D or using object of class D, then we must provide full-qualified name.

```cpp
// q7.cpp
#include<iostream>

using std::cout;

class B {
public:
    int i = 2;
    void print() {
        cout << "i = " << i << "\n";
    }
};

class D: public B {
    int i = 3;
public:
    void print(int x) {
        cout << "B::i = " << B::i << " x = " << x << "\n";
    }
};

int main() {
    B b;
    D d;

    b.print();
    d.print(3);

    return 0;
}
```

i = 2
B::i = 2 x = 3

```
// q8.cpp
#include<iostream>
class B {
protected:
    int i;
public:
    B(int i = 0) {
        this->i = i;
    }
};
class D: public B {
public:
    void print() {
        std::cout << i;
    }
    void print(B &b) {
        // error: ‘int B::i’ is protected within this context
        // Methods of class D can access
        // protected members of class B,
        // only if B object is part of object of D
        std::cout << b.i;
    }
};
int main() {
    D d;
    B b(5);
    d.print();
    d.print(d);
    return 0;
}
```

```
//q9.cpp
#include<iostream>
class B {
protected:
    int i;
public:
    B(int i = 0) {
        this->i = i;
    }
};
class D: public B {
public:
    D(int x): B(x) {
    }
    void print() {
        std::cout << i;
    }
    void print(D &d) {
        std::cout << d.i;
    }
};
int main() {
    D d1(3), d2(7);
    B b(5);
    d1.print();
    d1.print(d2);
    return 0;
}
```

- What is static type?
  - Type known at compile time
- What is dynamic type?
  - Type of the object that is being pointed/referred by pointer/reference. This is known at runtime.

```
class B {
};
class D: public B {
};
int main() {
    B *p;
    p = new D; // static type of p is B. But dynamic type of p is D
    return 0;
}
```

```
class B {
public:
    void f() {
        cout << "B::f()\n";
    }
};

class D: public B {
public:
    void f() {
        cout << "D::f()\n";
    }
};
```

```
int main() {

    B b;
    D d;

    B *b_p1 = &b;
    B *b_p2 = &d;

    B &b_r1 = b;
    B &b_r2 = d;

    b.f();          B::f()
    d.f();          D::f()
    b_p1->f();      B::f()
    b_p2->f();      B::f()
    b_r1.f();       B::f()
    b_r2.f();       B::f()

    return 0;
}
```

Examples of static binding

```
class B {
public:
    virtual void f() {
        cout << "B::f()\n";
    }
};

class D: public B {
public:
    void f() {
        cout << "D::f()\n";
    }
};
```

```
int main() {

    B b;
    D d;

    B *b_p1 = &b;
    B *b_p2 = &d;

    B &b_r1 = b;
    B &b_r2 = d;

    b.f();      B::f()      -- static binding
    d.f();      D::f()      -- static binding
    b_p1->f();  B::f()      -- dynamic binding
    b_p2->f();  D::f()      -- dynamic binding
    b_r1.f();   B::f()      -- dynamic binding
    b_r2.f();   D::f()      -- dynamic binding

    return 0;
}
```

- What is binding? - Choosing which method to execute for method call
- What is static binding?
  - a.k.a. early binding
  - Depends on static type
  - Method to be executed is decided at compile time
- What is dynamic binding? (Only possible when **virtual functions** are invoked)
  - a.k.a. late binding
  - Depends on dynamic type
  - Method to be executed is decided at runtime
  - Dynamic binding is done for **virtual function calls**
    - But sometimes compilers optimize it and decide it at compile time. (e.g. When Object is used to call the virtual method.) Though optimization will never change the expected output.
    - Exception being - explicit call to virtual function with full qualification
      - e.g. base_ptr->Base::virtual_fun(); // It will do static binding and will always call virtual_fun of Base, irrespective of type of object being pointed by base_ptr
  - Dynamic binding has nothing to do with dynamic memory. Dynamic binding is possible with stack and data segment as well (along with heap segment).

	Accessing method using **object**	Accessing method using **pointer** or **reference** to object
Non-virtual method	**Static binding**	**Static binding**
Virtual method	**Static binding**	**Dynamic binding**

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

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

```
int main() {

    B b;
    D d;

    B *b_p1 = &b;
    B *b_p2 = &d;

    B &b_r1 = b;
    B &b_r2 = d;

    b.f();          B::f()    -- static binding
    d.f();          D::f()    -- static binding
    b_p1->f();      B::f()    -- static binding
    b_p2->f();      B::f()    -- static binding
    b_r1.f();       B::f()    -- static binding
    b_r2.f();       B::f()    -- static binding
    b.g();          B::g()    -- static binding
    d.g();          D::g()    -- static binding
    b_p1->g();      B::g()    -- dynamic binding
    b_p2->g();      D::g()    -- dynamic binding
    b_r1.g();       B::g()    -- dynamic binding
    b_r2.g();       D::g()    -- dynamic binding

    return 0;
}
```

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

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

```
int main() {

    B b;
    D d;

    B *b_p1 = &b;
    B *b_p2 = &d;

    B &b_r1 = b;
    B &b_r2 = d;

    b.f();      B::f()   -- static binding
    d.f();      B::f()   -- static binding
    b_p1->f();  B::f()   -- static binding
    b_p2->f();  B::f()   -- static binding
    b_r1.f();   B::f()   -- static binding
    b_r2.f();   B::f()   -- static binding
    b.g();      B::g()   -- static binding
    d.g();      D::g()   -- static binding
    b_p1->g();  B::g()   -- dynamic binding
    b_p2->g();  D::g()   -- dynamic binding
    b_r1.g();   B::g()   -- dynamic binding
    b_r2.g();   D::g()   -- dynamic binding

    return 0;
}
```

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

class D: public B {
};
```

```
int main() {

    B b;
    D d;

    B *b_p1 = &b;
    B *b_p2 = &d;

    B &b_r1 = b;
    B &b_r2 = d;

    b.f();      B::f()    -- static binding
    d.f();      B::f()    -- static binding
    b_p1->f();  B::f()    -- static binding
    b_p2->f();  B::f()    -- static binding
    b_r1.f();   B::f()    -- static binding
    b_r2.f();   B::f()    -- static binding
    b.g();      B::g()    -- static binding
    d.g();      B::g()    -- static binding
    b_p1->g();  B::g()    -- dynamic binding
    b_p2->g();  B::g()    -- dynamic binding
    b_r1.g();   B::g()    -- dynamic binding
    b_r2.g();   B::g()    -- dynamic binding

    return 0;
}
```

```
class Animal {
public:
    virtual void sound() {
        cout << "Different animals have different sounds!\n";
    }
};

class Dog: public Animal {
public:
    virtual void sound() {
        cout << "Dog barks!\n";
    }
};

class Lion: public Animal {
public:
    virtual void sound() {
        cout << "Lion roars!\n";
    }
};
```

```
int main() {

    Animal animal, *animal_ptr = &animal;
    Dog dog;
    Lion lion;

    animal_ptr->sound();
    animal_ptr = &dog;
    animal_ptr->sound();
    animal_ptr = &lion;
    animal_ptr->sound();

    return 0;
}
```

```
Different animals have different sounds!
Dog barks!
Lion roars!
```

```
class Animal {
public:
    void sound() {
        cout << "Different animals have different sounds!\n";
    }
};

class Dog: public Animal {
public:
    virtual void sound() {
        cout << "Dog barks!\n";
    }
};

class Lion: public Animal {
public:
    virtual void sound() {
        cout << "Lion roars!\n";
    }
};
```

```
int main() {

    Animal animal, *animal_ptr = &animal;
    Dog dog;
    Lion lion;

    animal_ptr->sound();
    animal_ptr = &dog;
    animal_ptr->sound();
    animal_ptr = &lion;
    animal_ptr->sound();

    return 0;
}
```

```
Different animals have different sounds!
Different animals have different sounds!
Different animals have different sounds!
```

```
class Animal {
public:
    virtual void sound() {
        cout << "Different animals have different sounds!\n";
    }
};

class Dog: public Animal {
public:
    void sound() {
        cout << "Dog barks!\n";
    }
};

class Lion: public Animal {
public:
    void sound() {
        cout << "Lion roars!\n";
    }
};
```

```
int main() {

    Animal animal, *animal_ptr = &animal;
    Dog dog;
    Lion lion;

    animal_ptr->sound();
    animal_ptr = &dog;
    animal_ptr->sound();
    animal_ptr = &lion;
    animal_ptr->sound();

    return 0;
}
```

```
Different animals have different sounds!
Dog barks!
Lion roars!
```

```
class Animal {
public:
    virtual void sound() const {
        cout << "Different animals have different sounds!\n";
    }
};

class Dog: public Animal {
public:
    void sound() {
        cout << "Dog barks!\n";
    }
};

class Lion: public Animal {
public:
    void sound() {
        cout << "Lion roars!\n";
    }
};
```

```
int main() {

    Animal animal, *animal_ptr = &animal;
    Dog dog;
    Lion lion;

    animal_ptr->sound();
    animal_ptr = &dog;
    animal_ptr->sound();
    animal_ptr = &lion;
    animal_ptr->sound();

    return 0;
}
```

```
Different animals have different sounds!
Different animals have different sounds!
Different animals have different sounds!
```

- What is polymorphic type?
  - A class containing virtual member functions by definition or by inheritance is called polymorphic type
  - Polymorphic hierarchies are of great interest for software designers
- Constructor can not be virtual – why?
- Destructor must be virtual in polymorphic hierarchy – why?

```
class B {
public:
    int *p1= new int;
    B(int i) {
        *(this->p1) = i;
        cout << "B::B()" << endl;
    }
    ~B() {
        delete p1;
        cout << "B::~B()" << endl;
    }
    virtual void print() {
        cout << *p1 << endl;
    }
};
```

```
class D: public B {
public:
    int *p2= new int;
    D(int i, int j): B(i) {
        *(this->p2) = j;
        cout << "D::D()" << endl;
    }
    ~D() {
        delete p2;
        cout << "D::~D()" << endl;
    }
    void print() {
        cout << *p1 << " " << *p2 << endl;
    }
};
int main() {
    B *b_p = new D(1, 2);
    b_p->print();

    delete b_p;
    return 0;
}
```

```
B::B()
D::D()
1 2
B::~B()
```

```
class B {
public:
    int *p1= new int;
    B(int i) {
        *(this->p1) = i;
        cout << "B::B()" << endl;
    }
    virtual ~B() {
        delete p1;
        cout << "B::~B()" << endl;
    }
    virtual void print() {
        cout << *p1 << endl;
    }
};
```

```
class D: public B {
public:
    int *p2= new int;
    D(int i, int j): B(i) {
        *(this->p2) = j;
        cout << "D::D()" << endl;
    }
    ~D() {
        delete p2;
        cout << "D::~D()" << endl;
    }
    void print() {
        cout << *p1 << " " << *p2 << endl;
    }
};
int main() {
    B *b_p = new D(1, 2);
    b_p->print();

    delete b_p;
    return 0;
}
```

```
B::B()
D::D()
1 2
D::~D()
B::~B()
```

```
#define PI 3.14
class Shape {
public:
    virtual double area() = 0;
};
class Square: public Shape {
    double length;
public:
    Square(double length) {
        this->length = length;
    }
    double area() {
        return length * length;
    }
};
class Rectangle: public Shape {
    double length, width;
public:
    Rectangle(double length, double width) {
        this->length = length;
        this->width = width;
    }
    double area() {
        return length * width;
    }
};
```

```
class Circle: public Shape {
    double radius;
public:
    Circle(double radius) {
        this->radius = radius;
    }
    double area() {
        return PI * radius * radius;
    }
};
int main() {
    Shape *sp;
    sp = new Square(5.5);
    cout << sp->area() << endl;
    sp = new Rectangle(5.5, 6.6);
    cout << sp->area() << endl;
    sp = new Circle(5.5);
    cout << sp->area() << endl;
    return 0;
}
```

```
30.25
36.3
94.985
```

```
#define PI 3.14
class Shape {
public:
    virtual double area() = 0;
};
class Square: public Shape {
    double length;
public:
    Square(double length) {
        this->length = length;
    }
    double area() {
        return length * length;
    }
};
class Rectangle: public Shape {
    double length, width;
public:
    Rectangle(double length, double width) {
        this->length = length;
        this->width = width;
    }
    double area() {
        return length * width;
    }
};
```

```
class Circle: public Shape {
    double radius;
public:
    Circle(double radius) {
        this->radius = radius;
    }
    double area() {
        return PI * radius * radius;
    }
};
int main() {
    Shape *sp[3] = {
        new Square(5.5),
        new Rectangle(5.5, 6.6),
        new Circle(5.5)
    };
    for(int i = 0; i < 3; i++) {
        cout << sp[i]->area() << endl;
    }
    return 0;
}
```

```
30.25
36.3
94.985
```

- What is pure virtual function?
  - Function declaration in class definition ending with **= 0;**
  - Pure virtual function does not have body.
- What is abstract class?
  - Class with atleast one pure virtual function by inheritance or definition
  - Once function is defined in derived class it is no longer pure virtual function
  - Generally abstract class does not have state (data members) and all the methods are pure virtual functions
    - Though you can have mix of state, non-virtual, virtual and pure virtual functions in abstract class, it should be avoided.
  - Object can not be created for abstract class. And hence independently abstract class is not of much importance; unless it is used by other classes as base class. Hence almost always there will be a class deriving from abstract class. And hence it is known as Abstract Base Class (ABC)

```
//q10.cpp
#include<iostream>
using std::cout;
using std::endl;
class Parent {
public:
    virtual void abc() {
        cout << "abc" << endl;
    }
};
class Child: public Parent {
public:
    virtual void abc() {
        cout << "new abc" << endl;
    }
    virtual void xyz() {
        cout << "xyz" << endl;
    }
};
int main() {
    Parent *p;
    p = new Child;
    p->abc();
    //error: ‘class Parent’ has no member named ‘xyz’
    //p->xyz();
    return 0;
}
```

```
new abc
```

```
//q11.cpp
#include<iostream>
using std::cout;
using std::endl;
class Parent {
public:
    virtual void abc() {
        cout << "abc" << endl;
    }
};
class Child: public Parent {
public:
    virtual void abc() {
        cout << "new abc" << endl;
    }
    virtual void abc(int i) {
        cout << "new abc with int: " << i << endl;
    }
};
```

```
int main() {
    Parent *p;
    p = new Child;
    p->abc();
    //error: no matching function for call to ‘Parent::abc(int)’
    //p->abc(5);
    return 0;
}
```

```
new abc
```

```cpp
//q12.cpp
#include<iostream>
using std::cout;
using std::endl;
class Parent {
public:
    virtual void abc() {
        cout << "abc" << endl;
    }
    virtual void abc(int i) {
        cout << "abc with int: " << i << endl;
    }
};
class Child: public Parent {
public:
    virtual void abc() {
        cout << "new abc" << endl;
    }
    virtual void abc(int i) {
        cout << "new abc with int: " << i << endl;
    }
};
int main() {
    Parent *p;
    p = new Child;
    p->abc();
    p->abc(5);
    return 0;
}
```

```
new abc
new abc with int: 5
```

```
//q13.cpp
#include<iostream>
using std::cout;
using std::endl;
class Parent {
public:
    virtual void abc() {
        cout << "abc" << endl;
    }
    virtual void abc(int i) {
        cout << "abc with int: " << i << endl;
    }
};
class Child: public Parent {
public:
    virtual void abc() {
        cout << "new abc" << endl;
    }
};
```

```
int main() {
    Parent *p;
    p = new Child;
    p->abc();
    p->abc(5);
    return 0;
}
```

```
new abc
abc with int: 5
```

This works as pointer p has static type Parent for which abc(int) is not hidden.
Whether method is hidden or not depends on the static type of the pointer.

```cpp
//q14.cpp
#include<iostream>
using std::cout;
using std::endl;
class Parent {
public:
    virtual void abc() {
        cout << "abc" << endl;
    }
    virtual void abc(int i) {
        cout << "abc with int: " << i << endl;
    }
};
class Child: public Parent {
public:
    virtual void abc() {
        cout << "new abc" << endl;
    }
};
```

```cpp
int main() {
    Child *cp;
    cp = new Child;
    cp->abc();
    //error: no matching function for call to ‘Child::abc(int)’
    //cp->abc(5);
    return 0;
}
```

```
new abc
```

This does not work as pointer cp has static type Parent for which abc(int) is hidden.
Whether method is hidden or not depends on the static type of the pointer.

```
#include <iostream>
struct Base
{
    virtual void foo() {
        std::cout << "Base::foo()\n";
    }
    void bar() {
        foo();
    }
};
struct Derived : Base
{
    virtual void foo() {
        std::cout << "Derived::foo()\n";
    }
};
```

```
int main()
{
    Derived d;
    Base *b_p = &d;
    //d = *b_p;             //Error
    //Derived *d_p = b_p;   //Error
    (*b_p).foo(); // output: "Derived::foo()"
    b_p->foo();  // output: "Derived::foo()"
    d.bar();        // output: "Derived::foo()"
    b_p->bar();    // output: "Derived::foo()"
    d.Base::foo();  // output: "Base::foo()"
    return 0;
}
```

```
Derived::foo()
Derived::foo()
Derived::foo()
Derived::foo()
Base::foo()
```

foo() being called from bar() will also result in dynamic binding.
It is euivalent to **this->foo();**

Interesting reads

- *override* and *final* contextual keywords in C++
  - https://www.geeksforgeeks.org/override-keyword-c/
  - https://www.geeksforgeeks.org/c-final-specifier/
- How is dynamic binding achieved? - using Virtual Function Table
  - https://www.learncpp.com/cpp-tutorial/125-the-virtual-table/
- https://stackoverflow.com/questions/67821446/c-calling-inherited-virtual-method-using-derived-class-pointer-pointing-to-d
- https://stackoverflow.com/questions/1628768/why-does-an-overridden-function-in-the-derived-class-hide-other-overloads-of-the
- When virtual functions are invoked statically
  - https://stackoverflow.com/questions/43252822/when-virtual-functions-are-invoked-statically
- Why the dereference operator preserves polymorphism (late binding in c++)
  - https://stackoverflow.com/questions/23748057/why-the-dereference-operator-preserves-polymorphism-late-binding-in-c