SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220406<A|D>

Replace <A|D> with this section's letter

Copy Control, Multiple Inheritance, and Virtual Tables

—

CS 2124: Object Oriented Programming
Darryl Reeves, Ph.D.

Agenda

- Copy control
- Multiple inheritance
- Virtual tables
- End of Inheritance

Copy control

The Big 3

```
class SimpleClass {
public:
    SimpleClass() { p = new int(17); }

    SimpleClass(const SimpleClass& rhs) {
        p = new int;
        *p = *rhs.p;
    }

    SimpleClass& operator= (const SimpleClass& rhs) {
        if (this != &rhs) {
            delete p;
            p = new int;
            *p = *rhs.p;
        }
        return *this;
    }

    ~SimpleClass() { delete p; }

private:
    int* p;
};
```

copy constructor

assignment operator

destructor

Considerations when using inheritance

- copy control in derived class *independent* of base class's copy control

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }
};
```

```
int main() {
    cerr << "Derived der;\n"
         << "------------\n";
    Derived der;
}
```

```
% g++ -std=c++11 cc-inheritance.cpp -o cc-inheritance.o
% ./cc-inheritance.o
Derived der;
------------
Base()
Derived()
~Base()
```

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

};
```

```
int main() {
    cerr << "Derived der;\n"
         << "------------\n";
    Derived der;
}
```

```
% g++ -std=c++11 cc-inheritance.cpp -o cc-inheritance.o
% ./cc-inheritance.o
Derived der;
------------
Base()
Derived()
~Derived()
~Base()
```

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) {
        cerr << "Derived(const Derived&)\n";
    }
};
```

```
int main() {
    cerr << "Derived der;\n"
         << "------------\n";
    Derived der;

    cout << "===\n";
    cerr << "Derived der2(der);\n"
         << "------------------\n";
    Derived der2(der);
}
```

```
% g++ -std=c++11 cc-inheritance.cpp -o cc-inheritance.o
% ./cc-inheritance.o
...
===
Derived der2(der);
------------------
Base()
Derived(const Derived&)
~Derived()
~Base()
...
```

TurningPoint

SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220406<A|D>

Replace <A|D> with this section's letter

Where do we locate the call to `Base` class's copy constructor to ensure that it is the constructor invoked by `Derived` class's copy constructor?

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) {
        cerr << "Derived(const Derived&)\n";
    }

};
```

```
int main() {
    cerr << "Derived der;\n"
         << "------------\n";
    Derived der;

    cout << "===\n";
    cerr << "Derived der2(der);\n"
         << "------------------\n";
    Derived der2(der);
}
```

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) {
        cerr << "Derived(const Derived&)\n";
    }
};
```

```
int main() {
    cerr << "Derived der;\n"
         << "------------\n";
    Derived der;

    cout << "===\n";
    cerr << "Derived der2(der);\n"
         << "------------------\n";
    Derived der2(der);
}
```

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }
};
```

```
int main() {
    cerr << "Derived der;\n"
         << "------------\n";
    Derived der;

    cout << "===\n";
    cerr << "Derived der2(der);\n"
         << "------------------\n";
    Derived der2(der);
}
```

```
% g++ -std=c++11 cc-inheritance.cpp -o cc-inheritance.o
% ./cc-inheritance.o
...
===
Derived der2(der);
------------------
Base(const Base&)
Derived(const Derived&)
~Derived()
~Base()
...
```

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
};
```

```
int main() {
    cerr << "Derived der;\n"
         << "------------\n";
    Derived der;

    cout << "===\n";
    cerr << "Derived der2(der);\n"
         << "------------------\n";
    Derived der2(der);

    cout << "===\n";
    cerr << "Derived der3 = der;\n"
         << "-------------------\n";
    Derived der3 = der;
}
```

```
% g++ -std=c++11 cc-inheritance.cpp -o cc-inheritance.o
% ./cc-inheritance.o
...
===
Derived der3 = der;
-------------------
Base(const Base&)
Derived(const Derived&)
~Derived()
~Base()
...
```

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
};
```

```
int main() {
    cerr << "Derived der;\n"
         << "------------\n";
    Derived der;

    cout << "===\n";
    cerr << "Derived der2(der);\n"
         << "------------------\n";
    Derived der2(der);

    cout << "===\n";
    cerr << "der = der2;\n"
             << "-----------\n";
    der = der2;
}
```

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
};
```

```
int main() {
    cerr << "Derived der;\n"
         << "------------\n";
    Derived der;

    cout << "===\n";
    cerr << "Derived der2(der);\n"
         << "------------------\n";
    Derived der2(der);

    cout << "===\n";
    cerr << "der = der2;\n"
         << "-----------\n";
    der = der2;
}
```

```
% g++ -std=c++11 cc-inheritance.cpp -o cc-inheritance.o
% ./cc-inheritance.o
...
===
der = der2;
----------
Derived::operator=(const Derived&)
~Derived()
~Base()
...
```

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        ---
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
};
```

```
int main() {
    cerr << "Derived der;\n"
         << "------------\n";
    Derived der;

    cout << "===\n";
    cerr << "Derived der2(der);\n"
         << "------------------\n";
    Derived der2(der);

    cout << "===\n";
    cerr << "der = der2;\n"
         << "-----------\n";
    der = der2;
}
```

Which expression replaces blank #1 to invoke the `operator=()` method of the `Base` class?

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        _1_
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
};
```

```
int main() {
    cerr << "Derived der;\n"
         << "------------\n";
    Derived der;

    cout << "===\n";
    cerr << "Derived der2(der);\n"
         << "------------------\n";
    Derived der2(der);

    cout << "===\n";
    cerr << "der = der2;\n"
         << "-----------\n";
    der = der2;
}
```

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
};
```

```
int main() {
    cerr << "Derived der;\n"
         << "------------\n";
    Derived der;

    cout << "===\n";
    cerr << "Derived der2(der);\n"
         << "------------------\n";
    Derived der2(der);

    cout << "===\n";
    cerr << "der = der2;\n"
         << "-----------\n";
    der = der2;
}
```

```
% g++ -std=c++11 cc-inheritance.cpp -o cc-inheritance.o
% ./cc-inheritance.o
...
===
der = der2;
-----------
Base::operator=(const Base&)
Derived::operator=(const Derived&)
~Derived()
~Base()
...
```

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
};
```

```
int main() {
    cerr << "Derived* p = new Derived();\n"
         << "delete p;\n"
         << "--------------------------\n";
    Derived* p = new Derived();
    delete p;
}
```

```
% g++ -std=c++11 cc-inheritance.cpp -o cc-inheritance.o
% ./cc-inheritance.o
Derived* p = new Derived();
delete p;
--------------------------
Base()
Derived()
~Derived()
~Base()
```

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
};
```

```
int main() {
    cout << "===\n";
    cerr << "Base* bp = new Derived();\n"
         << "delete bp;\n"
         << "--------------------------\n";
    Base* bp = new Derived();
    delete bp;
}
```

```
% g++ -std=c++11 cc-inheritance.cpp -o cc-inheritance.o
% ./cc-inheritance.o
===
Base* bp = new Derived();
delete bp;
--------------------------
Base()
Derived()
~Base()
===
```

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    ___ ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
};
```

```
int main() {
    cout << "===\n";
    cerr << "Base* bp = new Derived();\n"
         << "delete bp;\n"
         << "--------------------------\n";
    Base* bp = new Derived();
    delete bp;
}
```

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    _2_ ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
};
```

```
int main() {
    cout << "===\n";
    cerr << "Base* bp = new Derived();\n"
         << "delete bp;\n"
         << "--------------------------\n";
    Base* bp = new Derived();
    delete bp;
}
```

Which keyword replaces blank #2 to ensure that the base class destructor can be overridden at runtime?

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    _2_ ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
};
```

```
int main() {
    cout << "===\n";
    cerr << "Base* bp = new Derived();\n"
         << "delete bp;\n"
         << "--------------------------\n";
    Base* bp = new Derived();
    delete bp;
}
```

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    virtual ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
};
```

```
int main() {
    cout << "===\n";
    cerr << "Base* bp = new Derived();\n"
         << "delete bp;\n"
         << "-------------------------\n";
    Base* bp = new Derived();
    delete bp;
}
```

```
% g++ -std=c++11 cc-inheritance.cpp -o cc-inheritance.o
% ./cc-inheritance.o
===
Base* bp = new Derived();
delete bp;
-------------------------
Base()
Derived()
~Derived()
~Base()
```

Considerations when using inheritance

- copy control in derived class *independent* of base class's copy control
- requirements when derived class implements copy control
  - derived class needs
    - copy constructor: call the Base copy constructor (in initialization list)
    - assignment operator: call the Base assignment operator (before doing anything else)
  - base class needs
    - destructor: mark it `virtual`

Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    virtual ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
};
```

```
int main() {
    cout << "===\n";
    cerr << "Base* bp = new Derived();\n"
         << "delete bp;\n"
         << "--------------------------\n";
    Base* bp = new Derived();
    delete bp;
}
```

Copy control with inheritance

```
class Member {
public:
    Member() {
        cerr << "Member()\n";
    }
    Member(const Member& rhs) {
        cerr << "Member(const Member&)\n";
    }
    Member& operator=(const Member& rhs) {
        cerr << "Member::operator=(const Member&)\n";
        return *this;
    }
    ~Member() {
        cerr << "~Member()\n";
    }
};
class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
private:
    Member mem;
};
```

Which methods of the `Derived` class need to be modified given the introduction of a member variable in the class definition?

```
class Member {
public:
    Member() {
        cerr << "Member()\n";
    }
    Member(const Member& rhs) {
        cerr << "Member(const Member&)\n";
    }
    Member& operator=(const Member& rhs) {
        cerr << "Member::operator=(const Member&)\n";
        return *this;
    }
    ~Member() {
        cerr << "~Member()\n";
    }
};
class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
private:
    Member mem;
};
```

Copy control with inheritance

```
class Member {
public:
    Member() {
        cerr << "Member()\n";
    }
    Member(const Member& rhs) {
        cerr << "Member(const Member&)\n";
    }
    Member& operator=(const Member& rhs) {
        cerr << "Member::operator=(const Member&)\n";
        return *this;
    }
    ~Member() {
        cerr << "~Member()\n";
    }
};
class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }
    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
private:
    Member mem;
};
```

Copy control with inheritance

```
class Member {
public:
    Member() {
        cerr << "Member()\n";
    }
    Member(const Member& rhs) {
        cerr << "Member(const Member&)\n";
    }
    Member& operator=(const Member& rhs) {
        cerr << "Member::operator=(const Member&)\n";
        return *this;
    }
    ~Member() {
        cerr << "~Member()\n";
    }
};
class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }
    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs), ___ {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
private:
    Member mem;
};
```

Copy control with inheritance

```
class Member {
public:
    Member() {
        cerr << "Member()\n";
    }
    Member(const Member& rhs) {
        cerr << "Member(const Member&)\n";
    }
    Member& operator=(const Member& rhs) {
        cerr << "Member::operator=(const Member&)\n";
        return *this;
    }
    ~Member() {
        cerr << "~Member()\n";
    }
};
class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }
    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs), _3_ {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
private:
    Member mem;
};
```

Which expression replaces blank #3 to initialize the member variable of the `Derived` class?

```
class Member {
public:
    Member() {
        cerr << "Member()\n";
    }
    Member(const Member& rhs) {
        cerr << "Member(const Member&)\n";
    }
    Member& operator=(const Member& rhs) {
        cerr << "Member::operator=(const Member&)\n";
        return *this;
    }
    ~Member() {
        cerr << "~Member()\n";
    }
};
class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }
    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs), _3_ {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
private:
    Member mem;
};
```

Copy control with inheritance

```
class Member {
public:
    Member() {
        cerr << "Member()\n";
    }
    Member(const Member& rhs) {
        cerr << "Member(const Member&)\n";
    }
    Member& operator=(const Member& rhs) {
        cerr << "Member::operator=(const Member&)\n";
        return *this;
    }
    ~Member() {
        cerr << "~Member()\n";
    }
};
class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }
    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs), mem(rhs.mem) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
private:
    Member mem;
};
```

Copy control with inheritance

```
class Member {
public:
    Member() {
        cerr << "Member()\n";
    }
    Member(const Member& rhs) {
        cerr << "Member(const Member&)\n";
    }
    Member& operator=(const Member& rhs) {
        cerr << "Member::operator=(const Member&)\n";
        return *this;
    }
    ~Member() {
        cerr << "~Member()\n";
    }
};
class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }
    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs), mem(rhs.mem) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        ---
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
private:
    Member mem;
};
```

Which statement replaces blank #4 to assign the appropriate value to the current object's `mem` variable?

```
class Member {
public:
    Member() {
        cerr << "Member()\n";
    }
    Member(const Member& rhs) {
        cerr << "Member(const Member&)\n";
    }
    Member& operator=(const Member& rhs) {
        cerr << "Member::operator=(const Member&)\n";
        return *this;
    }
    ~Member() {
        cerr << "~Member()\n";
    }
};
class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }
    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs), mem(rhs.mem) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        _4_
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
private:
    Member mem;
};
```

Copy control with inheritance

```
class Member {
public:
    Member() {
        cerr << "Member()\n";
    }
    Member(const Member& rhs) {
        cerr << "Member(const Member&)\n";
    }
    Member& operator=(const Member& rhs) {
        cerr << "Member::operator=(const Member&)\n";
        return *this;
    }
    ~Member() {
        cerr << "~Member()\n";
    }
};
class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }
    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs), mem(rhs.mem) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        mem = rhs.mem;
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
private:
    Member mem;
};
```

```
int main() {
    cout << "===\n";
    cerr << "Base* bp = new Derived();\n"
         << "delete bp;\n"
         << "-------------------------\n";
    Base* bp = new Derived();
    delete bp;
}
```

```
% g++ -std=c++11 cc-inheritance.cpp -o cc-inheritance.o
% ./cc-inheritance.o
===
Base* bp = new Derived();
delete bp;
-------------------------
Base()
Member()
Derived()
~Derived()
~Member()
~Base()
===
```

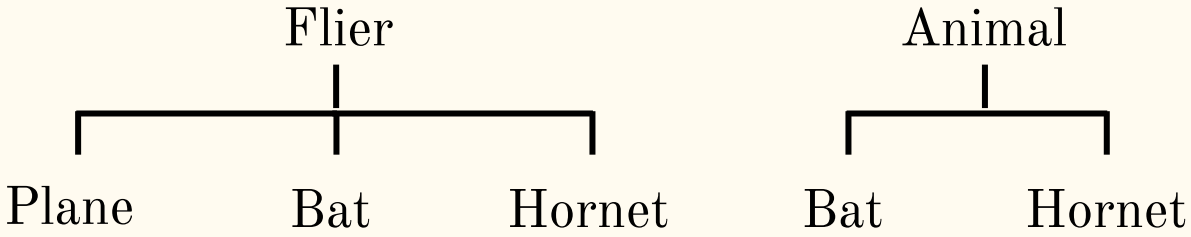
Copy control with inheritance

```
class Base {
public:
    Base() {
        cerr << "Base()\n";
    }
    Base(const Base& rhs) {
        cerr << "Base(const Base&)\n";
    }
    virtual ~Base() {
        cerr << "~Base()\n";
    }
    Base& operator=(const Base& rhs) {
        cerr << "Base::operator=(const Base&)\n";
        return *this;
    }
};

class Derived : public Base {
public:
    Derived() { cerr << "Derived()\n"; }

    ~Derived() {
        cerr << "~Derived()\n";
    }

    Derived(const Derived& rhs) : Base(rhs), mem(rhs.mem) {
        cerr << "Derived(const Derived&)\n";
    }

    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        cerr << "Derived::operator=(const Derived&)\n";
        return *this;
    }
};
```

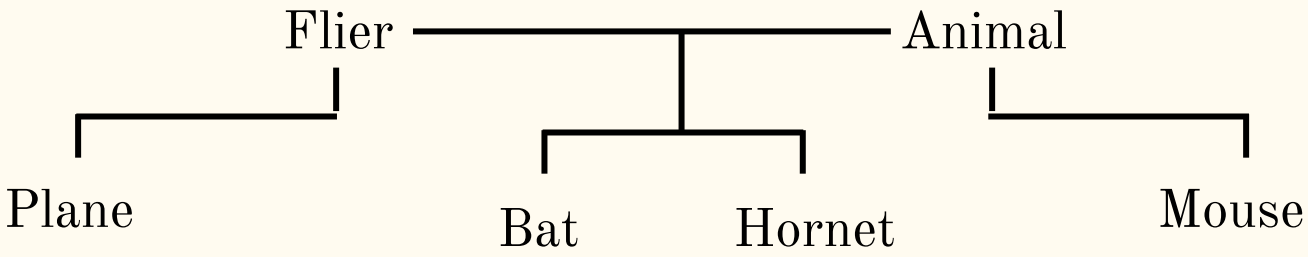
```
int main() {
    cerr << "Derived der;\n"
         << "------------\n";
    Derived der;
    cout << "===\n";
    cerr << "Derived der2(der);\n"
         << "-------------------\n";
    Derived der2(der);
    cout << "===\n";
    cerr << "Derived der3 = der;\n"
         << "--------------------\n";
    Derived der3 = der;
    cout << "===\n";
    cerr << "der = der2;\n"
         << "-----------\n";
    der = der2;
    cout << "===\n";
    cerr << "Derived* p = new Derived();\n"
         << "delete p;\n"
         << "---------------------------\n";
    Derived* p = new Derived();
    delete p;

    // Demonstrates need for virtual Base destructor
    cout << "===\n";
    cerr << "Base* bp = new Derived();\n"
         << "delete bp;\n"
         << "--------------------------\n";
    Base* bp = new Derived();
    delete bp;
    cout << "===\n";
}
```

Multiple inheritance

Inheriting from multiple classes



Inheriting from multiple classes



Supporting multiple inheritance

```
class Flier {
public:
    virtual void fly() { cout << "I can fly!!!\n"; }
};

class Animal {
public:
    virtual void display() { cout << "Animal\n"; }
};

class Bat : public Animal, public Flier { };

class Hornet : public Animal, public Flier {
public:
    void fly() {
        cout << "Bzzzz. ";
        Flier::fly();
    }
};

class Plane : public Flier {};
```

```
int main() {
    Bat battie;

    battie.display();
    battie.fly();
}
```

```
% g++ -std=c++11 mi.cpp -o mi.o
% ./mi.o
Animal
I can fly!!!
```

Supporting multiple inheritance

```
class Flier {
public:
    virtual void fly() { cout << "I can fly!!!\n"; }
};

class Animal {
public:
    virtual void display() { cout << "Animal\n"; }
};

class Bat : public Animal, public Flier { };

class Hornet : public Animal, public Flier {
public:
    void fly() {
        cout << "Bzzzz. ";
        Flier::fly();
    }
};

class Plane : public Flier {};
```

```
int main() {
    Bat battie;
    Plane canary;
    Hornet hugo;

    vector<Flier*> vf;
    vf.push_back(&battie);
    vf.push_back(&canary);
    vf.push_back(&hugo);

    for (Flier* flier : vf) {
        flier->fly();
    }
}
```

```
% g++ -std=c++11 mi.cpp -o mi.o
% ./mi.o
I can fly!!!
I can fly!!!
Bzzzz. I can fly!!!
```

Ambiguity in multiple inheritance

```
class Student {
public:
    virtual void display() const { cout << "Student\n"; }
};

class Instructor {
public:
    virtual void display() const { cout << "Instructor\n"; }
};

class TA : ___, ___ {
public:

};

int main() {
    TA sam;
    sam.display();
}
```

Ambiguity in multiple inheritance

```
class Student {
public:
    virtual void display() const { cout << "Student\n"; }
};

class Instructor {
public:
    virtual void display() const { cout << "Instructor\n"; }
};

class TA : _1_, ___ {
public:

};

int main() {
    TA sam;
    sam.display();
}
```

What replaces blank #1 to allow the `TA` class to inherit from the `Student` class?

```
class Student {
public:
    virtual void display() const { cout << "Student\n"; }
};

class Instructor {
public:
    virtual void display() const { cout << "Instructor\n"; }
};

class TA : _1_, ___ {
public:

};

int main() {
    TA sam;
    sam.display();
}
```

Ambiguity in multiple inheritance

```
class Student {
public:
    virtual void display() const { cout << "Student\n"; }
};

class Instructor {
public:
    virtual void display() const { cout << "Instructor\n"; }
};

class TA : public Student, ___ {
public:

};

int main() {
    TA sam;
    sam.display();
}
```

Ambiguity in multiple inheritance

```
class Student {
public:
    virtual void display() const { cout << "Student\n"; }
};

class Instructor {
public:
    virtual void display() const { cout << "Instructor\n"; }
};

class TA : public Student, _2_ {
public:

};

int main() {
    TA sam;
    sam.display();
}
```

What replaces blank #2 to allow the `TA` class to inherit from the `Instructor` class?

```
class Student {
public:
    virtual void display() const { cout << "Student\n"; }
};

class Instructor {
public:
    virtual void display() const { cout << "Instructor\n"; }
};

class TA : public Student, _2_ {
public:

};

int main() {
    TA sam;
    sam.display();
}
```

Ambiguity in multiple inheritance

```
class Student {
public:
    virtual void display() const { cout << "Student\n"; }
};

class Instructor {
public:
    virtual void display() const { cout << "Instructor\n"; }
};

class TA : public Student, public Instructor {
public:

};

int main() {
    TA sam;
    sam.display();
}
```

Ambiguity in multiple inheritance

```
class Student {
public:
    virtual void display() const { cout << "Student\n"; }
};

class Instructor {
public:
    virtual void display() const { cout << "Instructor\n"; }
};

class TA : public Student, public Instructor {
public:
    void display() const { cout << "TA\n"; }
};

int main() {
    TA sam;
    sam.display();
}
```

~~compilation error -- ambiguous~~

Virtual tables

Enabling dynamic binding

- Polymorphism allows for function definitions to be determined at runtime
- Enabled by associating a *virtual table* with each class with virtual function

Enabling dynamic binding

```
class A1 {};
class B1 : public A1 {};
class C1 : public B1 {};

class A2 {
public:
    void foo() {}
};
class B2 : public A2 {};
class C2 : public B2 {};

class A3 {
public:
    virtual void foo() {}
};
class B3 : public A3 { };
class C3 : public B3 { };
```

empty classes

inherited non-virtual function

inherited virtual function

Enabling dynamic binding

```
class A1 {};
class B1 : public A1 {};
class C1 : public B1 {};

class A2 {
public:
    void foo() {}
};
class B2 : public A2 {};
class C2 : public B2 {};

class A3 {
public:
    virtual void foo() {}
};
class B3 : public A3 { };
class C3 : public B3 { };
```

```
int main() {
    cout << "sizeof(A1): " << sizeof(A1)
         << ", sizeof(B1): " << sizeof(B1)
         << ", sizeof(C1) " << sizeof(C1) << endl;
}
```

```
% g++ -std=c++11 vtable.cpp -o vtable.o
% ./vtable.o
sizeof(A1): 1,  sizeof(B1): 1, sizeof(C1) 1
```

Enabling dynamic binding

```
class A1 {};
class B1 : public A1 {};
class C1 : public B1 {};

class A2 {
public:
    void foo() {}
};
class B2 : public A2 {};
class C2 : public B2 {};

class A3 {
public:
    virtual void foo() {}
};
class B3 : public A3 { };
class C3 : public B3 { };
```

```
int main() {
    cout << "sizeof(A1): " << sizeof(A1)
         << ",  sizeof(B1): " << sizeof(B1)
         << ", sizeof(C1) " << sizeof(C1) << endl;

    cout << "sizeof(A2): " << sizeof(A2)
         << ",  sizeof(B2): " << sizeof(B2)
         << ", sizeof(C2) " << sizeof(C2) << endl;
}
```

```
% g++ -std=c++11 vtable.cpp -o vtable.o
% ./vtable.o
sizeof(A1): 1,  sizeof(B1): 1, sizeof(C1) 1
sizeof(A2): 1,  sizeof(B2): 1, sizeof(C2) 1
```

Enabling dynamic binding

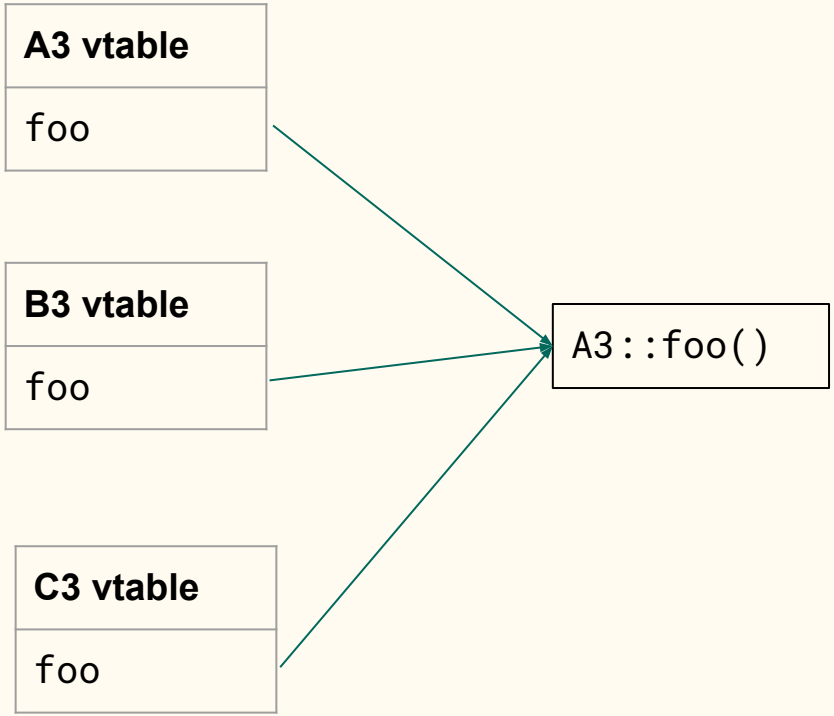
```
class A1 {};
class B1 : public A1 {};
class C1 : public B1 {};

class A2 {
public:
    void foo() {}
};
class B2 : public A2 {};
class C2 : public B2 {};

class A3 {
public:
    virtual void foo() {}
};
class B3 : public A3 { };
class C3 : public B3 { };
```

```
int main() {
    cout << "sizeof(A1): " << sizeof(A1)
         << ",  sizeof(B1): " << sizeof(B1)
         << ", sizeof(C1) " << sizeof(C1) << endl;

    cout << "sizeof(A2): " << sizeof(A2)
         << ", sizeof(B2): " << sizeof(B2)
         << ", sizeof(C2) " << sizeof(C2) << endl;

    cout << "sizeof(A3): " << sizeof(A3)
         << ", sizeof(B3): " << sizeof(B3)
         << ", sizeof(C3) " << sizeof(C3) << endl;
}
```

```
% g++ -std=c++11 vtable.cpp -o vtable.o
% ./vtable.o
sizeof(A1): 1,  sizeof(B1): 1, sizeof(C1) 1
sizeof(A2): 1,  sizeof(B2): 1, sizeof(C2) 1
sizeof(A3): 8,  sizeof(B3): 8, sizeof(C3) 8
```

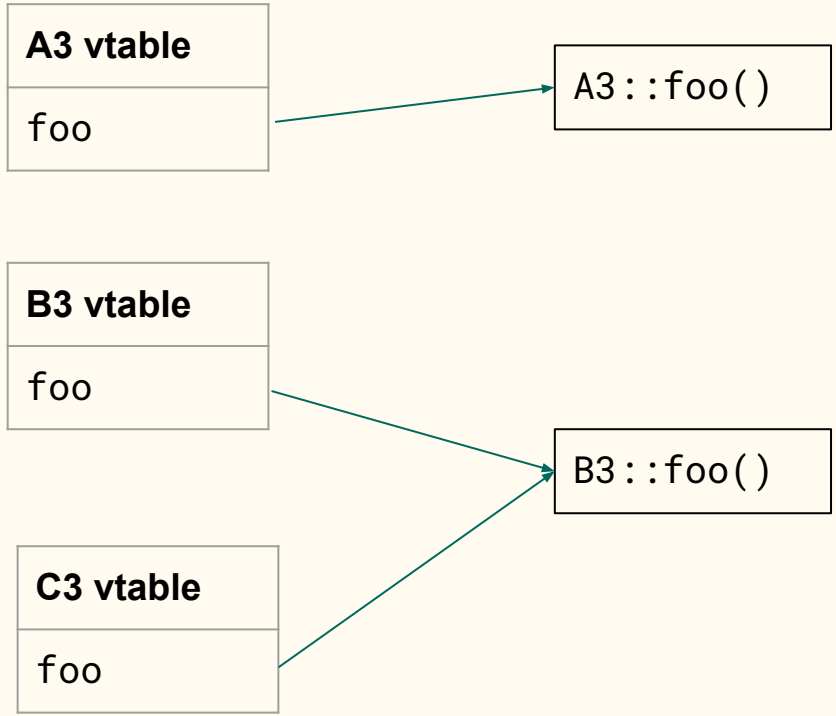
Enabling dynamic binding

```
class A3 {
public:
    virtual void foo() {}
};
class B3 : public A3 { };
class C3 : public B3 { };
```



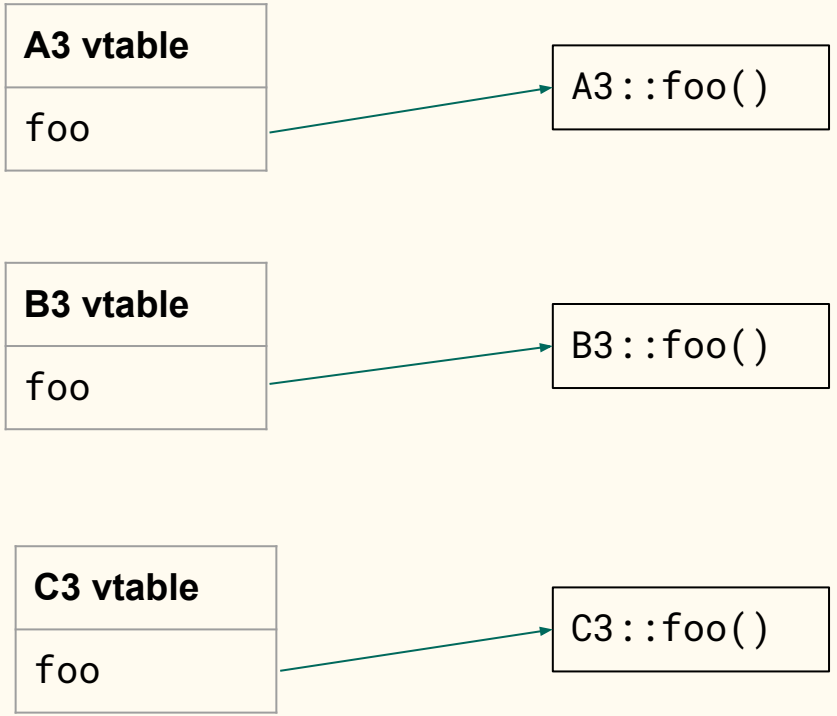
Enabling dynamic binding

```
class A3 {
public:
    virtual void foo() {}
};
class B3 : public A3 {
public:
    void foo() {}
};
class C3 : public B3 { };
```



Enabling dynamic binding

```
class A3 {
public:
    virtual void foo() {}
};
class B3 : public A3 {
public:
    void foo() {}
};
class C3 : public B3 {
public:
    void foo() {}
};
```



Final thoughts on Inheritance

Inheritance assignment rules

- derived class instance to base class instance ✔
- base class instance to derived class instance ✘
- address of derived class instance to base class pointer ✔
- address of base class instance to derived class pointer ✘

All are compile time considerations

Polymorphism

- Derived classes can be used in place of a base class

```
class Animal {};

class Lion : public Animal {};

class Tiger : public Animal {};

class Bear : public Animal {};
```

any where an Animal is expected, a Lion, Tiger, or Bear can be provided at runtime

Declared type vs actual type

- Compiler evaluates code based on declared type
- Actual type provided at **runtime** can be of a derived class

Polymorphism and function parameters

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};

class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};

void feed_animal(Animal& an) {
    cout << "Feeding the animal\n";
    an.eat();
}
```

```
int main() {
    Lion leo;

    feed_animal(leo);
}
```

can pass Animal, Lion, or Bear instance at runtime

```
Feeding the animal
Lion eating
```

eat() must be defined for Animal class for code to compile

Polymorphism and function parameters

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
    void climb() { cout << "Lion climbing\n"; }
};

class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
void feed_animal(Animal& an) {
    cout << "Feeding the animal\n";
    an.eat();
}
```

```
int main() {
    Animal* an_ptr = new Lion();

    an_ptr->climb();   compilation error!
}
```