IDK1531 Advanced C++ Course

Inheritance
Dynamic binding

Aleksandr Lenin

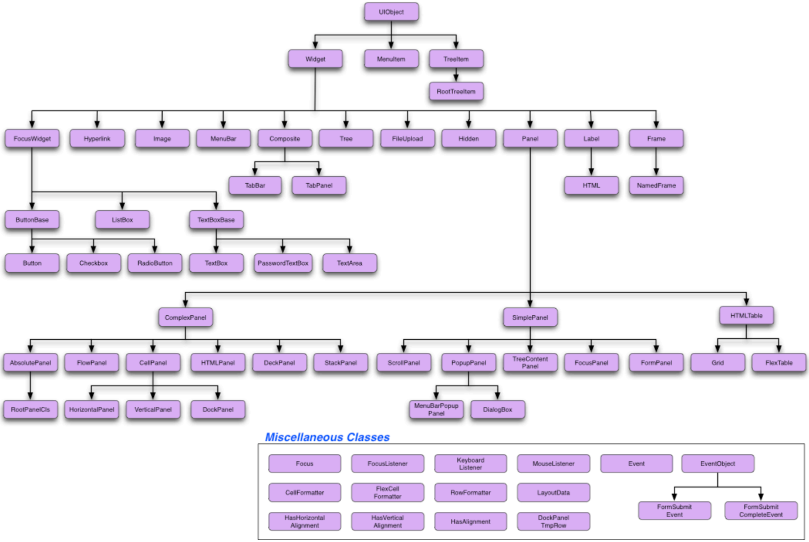
April 23rd, 2019

The *key ideas* of object-oriented programming are:

1. **data abstraction** – classes separate interface from implementation.
2. **incapsulation** – objects are self-contained self-sufficient entities that have *states*.
3. **inheritance** – model relationships among similar types.
4. **dynamic binding** – use objects ignoring the details of how they differ.

Inheritance:

- Classes related by **inheritance** form a hierarchy
- There is a **base class** at the root of the hierarchy
- Other classes inherit from the base class, directly or indirectly
- Inheriting classes are known as **derived classes**
- Base class defines members that are common to the types in the hierarchy
- Derived class defines members that are specific to the derived class itself
- Base class distinguishes between:
    - Type-dependent functions – derived classes define by themselves (defined as **virtual functions**)
    - Type-independent functions – derived classes inherit without change


UIObject
Widget
MenuItem
TreeItem
RootTreeItem
FocusWidget
Hyperlink
Image
MenuBar
Composite
Tree
FileUpload
Hidden
Panel
Label
Frame
TabBar
TabPanel
HTML
NamedFrame
ButtonBase
ListBox
TextBoxBase
Button
Checkbox
RadioButton
TextBox
PasswordTextBox
TextArea
ComplexPanel
SimplePanel
HTMLTable
AbsolutePanel
FlowPanel
CellPanel
HTMLPanel
DeckPanel
StackPanel
ScrollPanel
PopupPanel
TreeContent Panel
FocusPanel
FormPanel
Grid
FlexTable
RootPanelCls
HorizontalPanel
VerticalPanel
DockPanel
MenuBarPopup Panel
DialogBox
Miscellaneous Classes
Focus
FocusListener
Keyboard Listener
MouseListener
Event
EventObject
CellFormatter
FlexCell Formatter
RowFormatter
LayoutData
FormSubmit Event
FormSubmit CompleteEvent
HasHorizontal Alignment
HasVertical Alignment
HasAlignment
DockPanel TmpRow

A derived class must specify the class(es) from which it intends to inherit in a class derivation list – a colon (:) followed by a comma-separated list of base classes, each of which may have an optional access specifier

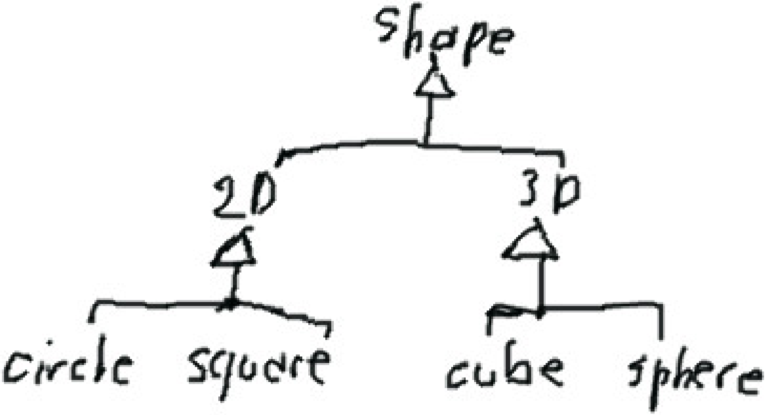
```
class Base {
    virtual void doSomething();
};

class Deriv : public Base {
    void doSomething() override;
};
```

`: public Base` means that we can use objects of type `Deriv` *as if* they were `Base` objects

- The derived class may provide an implementation of the virtual functions it intends to re-define
- The derived class may include the `virtual` keyword on these functions, but is not required to do so
- In order to explicitly state that it intends to override a virtual function that it inherits, the `override` keyword is added to the function declaration after its parameter list
- *Single inheritance* – classes inherit directly from only one base class
- *Multiple inheritance* – classes inherit from multiple base classes

Let us start with a trivial example:



```
class Shape {
public:
    Shape() = default;
    Shape(string color)
        : _color(color) {}
    virtual ~Shape() = default;
    virtual string getTypeInfo(){
        return string("Some shape"); }
    string getColor() { return _color; }
protected:
    string _color;
};
```

Classes used as the root of an inheritance hierarchy *almost always* define a virtual destructor.

```
class Shape2D : public Shape {
public:
    Shape2D(string c) : Shape(c) {}
    string getTypeInfo() override {
        return string("2D Shape");
    }
protected:
    Vector2D origin;
};
```

Access modes:

- **public** – accessible to derived classes, from outside of the class, to friends of the class
- **private** – inaccessible to derived classes nor from outside of the class (accessible only within the class itself)
- **protected** – accessible to derived classes and friends of the class, inaccessible from outside of the class. A derived class member of a friend may access the protected members of the base class only via the derived object – there is no direct access to the protected members of the base class (consider the example on the next slide)

```
class Base {
protected:
    int prot;
};
class Derived : public Base {
    friend void foo(Base&);
    friend void foo(Derived&);
};
foo(Derived& d) { d.prot = 0; }  // ok
foo(Base& b) { b.prot = 0; }     // error
```

Public inheritance `class Deriv : public Base`:

- public → public
- protected → protected
- Additionally, we can bind an object of a publicly derived type to a pointer of a reference to the base type

Private inheritance `class Deriv : private Base`:

- public → private
- protected → private

Protected inheritance `class Deriv : protected Base`:

- public → protected
- protected → protected

Consider the following class definition

```
class A {
public:
    int a,b,c,d,e,f,g,h;
protected:
    int x,y,z;
};
```

We need to achieve the state in which in the derived class `B`, the variables `a` and `x` have `public` access, variables `y` and `z` have `protected` access, all other variables have `private` access.

Solution:

```
class B : private A {
public:
    using A::a;
    using A::x;
protected:
    using A::y;
    using A::z;
};
```

Result:

- `a` and `x` are `public`
- `y` and `z` are `protected`
- `b`,`c`,`d`,`e`,`f`,`g`,`h` are `private`

```cpp
class Square : public Shape2D {
public:
    Square(string c="black", int w=0,
        int h=0) : Shape2D(c), width(w),
        height(h) {}
    string getTypeInfo() override {
        return string("Square");
    }
    int area() { return width*height; }
protected:
    int width, height;
};
```

Virtual Functions

Virtual functions:

- The base class defines as **virtual** those functions that it expects its derived classes to *override*
- If a virtual function is called using either *a pointer or a reference*, the call is *dynamically bound*
- A base class specifies that a member function should be dynamically bound by preceding its declaration with the keyword `virtual`
- Any non-static function, other than a constructor, may be virtual
- The `virtual` keyword appears only within the declaration inside the class
- The `virtual` keyword may not be used in function definition outside the class body

Virtual functions:

- A function that is declared as `virtual` in the base class is implicitly `virtual` in the derived classes
- Derived classes frequently (but not always) override the virtual functions that they inherit
- If a derived class does not override an inherited virtual function, the derived class inherits the version defined in its base class
- A derived class may include `virtual` keyword on the functions it overrides, but is not required to do so
- In order to explicitly note that the derived class intends a member function to override an inherited virtual function, the `override` keyword should be added after the parameter list or if the member is `const`, after the `const` or reference qualifiers
- Virtual functions are resolved during run-time

Dynamic Binding

Dynamic binding:

- When a *virtual function* is called using a pointer or a reference, the call will be *dynamically bound*
- Depending on the type of the object, to which the reference or pointer is bound, the function version in the base class or in one of its derived classes will be executed

Calling virtual function by reference:

```
Shape2D s2("white");
Square s3("red",10,10);
Shape s, &ref1=s2, &ref2=s3;
cout << s.getTypeInfo() << endl
     << ref1.getTypeInfo() << endl
     << ref2.getTypeInfo() << endl;
```

Output:

```
Some shape
2D Shape
Square
```

Same, but using *rvalue* initialization:

```
Shape2D s2("white");
Square s3("red",10,10);
Shape s, &&ref1 = Shape2D("white"),
    &&ref2 = Square("red",10,10);
cout << s.getTypeInfo() << endl
     << ref1.getTypeInfo() << endl
     << ref2.getTypeInfo() << endl;
```

Output:

```
Some shape
2D Shape
Square
```

Using pointers bound to the base class:

```
unique_ptr<Shape> p1(new Shape());
unique_ptr<Shape> p2(new Shape2D("white"));
unique_ptr<Shape> p3(
    new Square("red",10,10));
cout << p1->getTypeInfo() << endl
     << p2->getTypeInfo() << endl
     << p3->getTypeInfo() << endl;
```

Output:

```
Some shape
2D Shape
Square
```

Static members and inheritance:

- If a base class defines a static member, there is only one such member defined for the entire class hierarchy
- Regardless of the number of classes derived from a base class, there exists a single instance of each static member
- Static members obey normal access controls

```
class Base {
public:
    static void statmem();
};

class Derived : public Base {
    void f(const Derived&);
};
```

We can prevent a class from being used as a base by following the class name with the `final` keyword:

```
class Square final : public Shape2D {
public:
    Square(string c="black", int w=0,
        int h=0) : Shape2D(c), width(w),
        height(h) {}
    string getTypeInfo() override {
        return string("Square");
    }
    int area() { return width*height; }
protected:
    int width, height;
};
```

We can also designate a function as `final`. Any attempt to override a function that has been defined as `final` will be flagged as an error

```
class Square final : public Shape2D {
public:
    Square(string c="black", int w=0,
        int h=0) : Shape2D(c), width(w),
        height(h) {}
    string getTypeInfo() override {
        return string("Square");
    }
    int area() final { return width*height; }
protected:
    int width, height;
};
```

In some cases, we want to prevent dynamic binding of a call to a virtual function, we want to force the call to use a particular version of that virtual. We can use the scope operator to do so. For example, consider this code:

```
unique_ptr<Square> ptr(
    new Square("green",10,10));
cout << ptr->Shape::getTypeInfo() << endl
     << ptr->Shape2D::getTypeInfo() << endl
     << ptr->Square::getTypeInfo() << endl
     << ptr->getTypeInfo() << endl;
```

Why might we wish to circumvent the virtual mechanism? The most common reason is when a derived-class virtual function calls the version from the base class.

Abstract Class

An *abstract class* is the type that:

- cannot be instantiated
- can be used as the base class
- defines or inherits at least one method marked with pure virtual modifier (`= 0`)
- the destructor can be marked pure virtual too!
  `~Foo() = 0;`
- If a derived class inherits a pure virtual method, it becomes an abstract class as well, unless it overrides the method and provides its implementation

Interface

Interfaces:

- An *interface* consists of *pure virtual functions*
- Each class implementing an interface must override its pure virtual methods
- There is no way to create an object of an interface type, as an interface is an *abstract base class*
- Interfaces specify API call conventions

```
class Interface {
public:
    virtual int foo() = 0;
    virtual double bar() = 0;
    virtual string baz() const = 0;
};

class Impl : public Interface {
public:
    int foo() override {  return 0; }
    double bar() override { return 0.0; }
    string baz() const override { return ""; }
};
```

Derived to Base Conversion

The *derived to base* conversion is used to enable dynamic binding

User code may use it *only* in the case of *public inheritance*

If a public member of the base class would be accessible, then the derived to base conversion is also accessible, not otherwise

```
class A {};
class B : public A {};
class C : protected A{};
class D : private A{};

int main() {
    A *b = new B();  // ok

    // 'A' is an inaccessible base of 'C'
    A *c = new C();

    // 'A' is an inaccessible base of 'D'
    A *d = new D();

    return 0;
}
```

```
class B : public A {
public:
    void foo(const A &a) {}
};

int main() {

    // ok, due to public inheritance
    B b,c; b.foo(c);

    return 0;
}
```

```
class B : protected A {
public:
    void foo(const A &a) {}
};

int main() {

    // error: 'A' is an
    // inaccessible base of 'B'
    B b,c; b.foo(c);

    return 0;
}
```

```
class B : private A {
public:
    void foo(const A &a) {}
};

int main() {

    // error: 'A' is an
    // inaccessible base of 'B'
    B b,c; b.foo(c);

    return 0;
}
```

The *derived to base* conversion to a *direct base class* is always accessible to members and friends of the derived class.

```
class C : protected A {
public:
    void foo(C *other) { A *a = other; }
};
class D : private A {
public:
    void foo(D *other) { A *a = other; }
};

int main() {

    C *c = new C(); c->foo(new C());
    D *d = new D(); d->foo(new D());

    // A is inaccesisble base of C
    A *a = new C();
    return 0;
}
```

Member functions and friends of classes derived from `B` may use the *derived to base* conversion if `B` inherits from `A` using either `public` or `protected` inheritance.

```
class A {};

class B : public A {};

class C : public B {
public:

    // ok
    // because B inherits A in a public way
    void foo(C *other) { A *a = other; }
};
```

```
class A {};

class B : protected A {};

class C : public B {
public:

    // ok
    // because B inherits A in a protected way
    void foo(C *other) { A *a = other; }
};
```

```cpp
class A {};

class B : private A {};

class C : public B {
public:

    // error: 'A' is an inaccessible base of 'C'
    // because B inherits A in a private manner
    void foo(C *other) { A *a = other; }
};
```

Friendship & Inheritance

Friendship is not inherited

Friends of the base class have no special access to members of its derived class

Friends of the derived class have no special access to the base class

```
class Base {
    friend class Pal;
public:
    Base(int priv, int prot)
        : base_priv(priv), base_prot(prot) {}
private:
    int base_priv;
protected:
    int base_prot;
};

class Base2 {
private:
    int base2_priv;
protected:
    int base2_prot;
};
```

Each class controls access to its own members

```
class Pal : protected Base {
public:
    Pal(int priv, int prot) : Base(priv,prot) {

    // ok, Pal is a friend
    int f(Base b) { return b.base_priv; }

    // error: 'int Base2::base2_priv'
    // is private because Pal is not
    // a friend of Base2
    void f2(Base2 b) { return b.base2_priv; }

};
```

Class `D` has no access to protected and private members in `Base`

```
class D : public Pal {
public:
    // error: 'int Base::base_prot'
    // is protected
    int mem(Base b) { return b.base_prot; }
};
```

a remark on

”Inheriting Constructors”

- Introduced in C++11, improved in C++14
- Invalid and misleading term used in literature
- Constructors are NOT inherited!

```
struct Base {
    Base(int a) : i(a) {}
    int i;
};

struct Derived : Base {
    Derived(int a, std::string s) : Base(a), m(s) {}

    using Base::Base;
    // Inherit Base's constructors. Equivalent to:
    // Derived(int a) : Base(a), m() {}

    std::string m;
};
```

Multiple Inheritance

Multiple inheritance – ability to derive a class from more than one direct base class

A multiply derived class inherits properties of all its parents

Tricky design-level and implementation-level problems

```
class Duck : public Bird, public Swimming,
public Flying {};

class Penguin : public Bird,
public Swimming {};

class Chicken : public Bird {};

class Cockoo : public Bird, public Flying {};
```

Constructing an object of the derived type constructs and initializes all its base sub-objects

The order in which base classes are constructed depends on the order in which they are listed in the class derivation list

```
class Duck : public Bird, public Swimming,
public Flying {
    public:
    Duck() : Bird(), Swimming(), Flying() {}
};
```

A class that inherits the same constructor from more than one base class must define its own version of that constructor

```
class Deriv : public Base1, public Base2 {
public:
    using Base1::Base1; // inherit
    using Base2::Base2; // inherit
    Deriv(const std::string& s)  :
    public Base1, public Base2 {};
    Deriv() = default;
};
```

Destructor in derived class is responsible for cleaning up the resources allocated by that class only

The members of all base classes of the derived class are automatically destroyed

Destructors are always invoked in the reverse order from which the constructors are run. In this case `~Flying(),~Swimming(),~Bird()`

A pointer or a reference to any of an object's (accessible) base classes can be used to point or refer to a derived object

A class can inherit from the same base class more than once – the diamond problem

This can be achieved via inheriting the same base class indirectly from its direct base classes

For instance:

`iostram --> { istream, ostream } --> basic_ios`

`basic_ios` is inherited twice, but `iostream` wants to use the same buffer for IO operations

Virtual inheritance lets a class specify that it is willing to share its base class

The shared base class sub-object is called a *virtual base class*

Derived object contains *only one* single object for that virtual base class

```
class Base {};
class A : public virtual Base {};
class B : virtual public Base {};
class Deriv : public A, public B {};
```

```
class Person {
public:
    Person(int x)  {
        std::cout << "Person::Person(int) called"
                  << std::endl;
    }
};

class Faculty : public Person {
public:
    Faculty(int x):Person(x)   {
        std::cout << "Faculty::Faculty(int) "
                  << "called" << std::endl;
    }
};
```

```
class Student : public Person {
public:
    Student(int x):Person(x) {
        std::cout << "Student::Student(int) "
                  << "called" << std::endl;
    }
};

class TA : public Faculty, public Student
{
public:
    TA(int x): Student(x), Faculty(x)   {
        std::cout << "TA::TA(int) called"
                  << std::endl;
        }
};
```

```
int main()  {
    TA ta1(30);
}
```

Output:

```
Person::Person(int) called
Faculty::Faculty(int) called
Person::Person(int) called
Student::Student(int) called
TA::TA(int) called
```

Constructor of `Person` is called two times.

Destructor of `Person` will be called two times when object `ta1` is destructed.

Object `ta1` has two copies of all members of `Person`, this causes ambiguities.

Solution:

Make the classes `Faculty` and `Student` as virtual base classes to avoid two copies of `Person` in `TA` class.

```
class Person {
public:
    Person(int x)  {
        std::cout << "Person::Person(int) "
                  << "called" << std::endl;
    }

    Person() {
        std::cout << "Person::Person() "
                  << "called" << std::endl;
    }
};
```

```
class Faculty : public virtual Person {
public:
    Faculty(int x):Person(x)   {
        std::cout << "Faculty::Faculty(int) "
                  << "called" << std::endl;
    }
};

class Student : public virtual Person {
public:
    Student(int x):Person(x) {
        std::cout << "Student::Student(int) "
                  << "called" << std::endl;
    }
};
```

```
class TA : public Faculty, public Student {
public:
    TA(int x) : Student(x), Faculty(x)  {
        std::cout << "TA::TA(int) called"
                  << std::endl;
    }
};

int main() {
    TA ta1(30);
}
```

Output:

```
Person::Person() called
Faculty::Faculty(int) called
Student::Student(int) called
TA::TA(int) called
```

```
Person::Person() called   <----- NOTICE THAT!
Faculty::Faculty(int) called
Student::Student(int) called
TA::TA(int) called
```

When `virtual` inheritance is used, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

How to call the parametrized constructor of the ***`Person`*** *class?*

The constructor has to be called in `TA` class.

```
class TA : public Faculty, public Student
{
public:
    TA(int x) : Student(x), Faculty(x),
    Person(x)   {

        std::cout << "TA::TA(int) called"
                  << std::endl;
    }
};
```

Output:

```
Person::Person(int) called   <--- SEE?
Faculty::Faculty(int) called
Student::Student(int) called
TA::TA(int) called
```

REMARK

In general, it is not allowed to call the grandparent's constructor directly, it has to be called through parent class. It is allowed only in the case of virtual inheritance.

Cross–Delegation

aka delegation to sister class

```
class Base {
public:
    virtual void foo() = 0;
    virtual void bar() = 0;
};

class A : public virtual Base {
public:
    void foo() override;
};

class B : public virtual Base {
public:
    void bar() override;
};

void A::foo() { this->bar(); }
```

```
void B::bar() {
    std::cout << "B::bar() is called"
              << std::endl;
}

class C : public A, public B {};

int main() {
    (new C())->foo();
    return 0;
}
```

Output:

```
B::bar() is called
```

What did just happen?

When `A::foo()` calls `this->bar()`, it ends up calling `B::bar()`.

This way, a class, that `A` knows nothing about, supplies the override of a virtual function invoked by `A::foo()`. This becomes possible due to virtual inheritance.

?

THANK YOU
FOR
YOUR
ATTENTION
ANY QUESTIONS?