CIS 330
C++ and Unix

Lecture 14

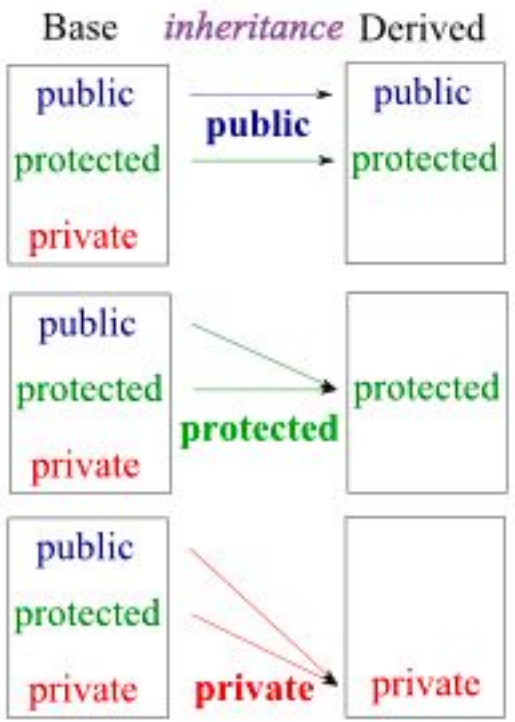
C++ - Inheritance and Composition

Composition

- Composition – Combining data types to create new classes
  - Most classes are made from compositions (i.e., from built-in types)
  - User-defined types act similarly to build-in types

Inheritance

- class <derived class> : <access level> <base class> { }



What is Inherited

- Everything but…
  - Constructors and destructors
  - Assignment operator (operator=, covered later in operator overloading)
  - Friends
  - Private members
- Constructors and destructors
  - Not inherited, but automatically called by the constructors and destructors of the derived class (otherwise you won't be able to create the base class)
  - Default is called, but you can specify other constructors using the **initialization list**
  - `Tree(int treeHeight) : height(treeHeight) {}`
- You can inherit from more than one class

Example

```
1.  class Mother {
2.  protected:
3.      int i;
4.  public:
5.      Mother() {
6.          cout << "Mother constructor without arguments" << endl; }
7.      Mother(int a) {
8.          cout << "Mother constructor with int argument" << endl;
9.          i = a; }
10. };
11. class Daughter : public Mother {
12. public:
13.     Daughter(int a) {
14.         cout << "Daughter: int parameter\n\n"; }
15.     void print() { cout << i << endl; }
16. };
20. class Son : public Mother {
21. public:
22.     Son(int a) : Mother(a) {
23.         cout << "Son: int parameter\n\n"; }
24.     void print() { cout << i << endl; }
25. };
26.     Daughter Jane(0);
27.     Son John(0);
28.     Jane.print();
29.     John.print();
```

Example

```
Mother constructor without arguments
Daughter: int parameter


Mother constructor with int argument
Son: int parameter


1520081968
0
```

Inheritance

- In inheritance, the derived class still has all elements of the base class (**including the private ones**).
- However, the ones that it does not have access to (e.g., private items) cannot be accessed even if it exists
- If you compare the size of the two classes (i.e., using sizeof()), you will see that size of the base class is smaller than that of the inherited class

Example

```
class someBaseClass {
private:
    int i;
};
class someDerivedClass : public someBaseClass {
private:
    int i;
};
cout << "base: " << sizeof(someBaseClass) << endl;
cout << "derived: " << sizeof(someDerivedClass) << endl;
```

Example

```
base: 4
derived: 8
```

Inheritance

- If an inherited class redefines a base class's function, subsequent calls will refer to the inherited class's version (if you don't like the behavior of a certain function that you inherited, you can change its behavior)
- You can still call the base class's function by specifying the **class name and ::**

Inheritance

```
class B {
private:
    int i;
public:
    B() { i = 0; }
    int f() { return i; }
    int h() { return i + 10; }
};

class D : public B {
private:
    int i;
public:
    D() { i = 1; }
    int g() { return i; }
    int h() { return i + 5; }
};
```

```
D d;
cout << d.g() << endl;
cout << d.f() << endl;
```

Inheritance

```
class B {
private:
    int i;
public:
    B() { i = 0; }
    int f() { return i; }
    int h() { return i + 10; }
};


class D : public B {
private:
    int i;
public:
    D() { i = 1; }
    int g() { return i; }
    int h() { return i + 5; }
};
```

It will return 1 from the inherited class

```
D d;
cout << d.g() << endl;
cout << d.f() << endl;
```

It will return 0 from the base class

Inheritance

```
class B {
private:
    int i;
public:
    B() { i = 0; }
    int f() { return i; }
    int h() { return i + 10; }
};

class D : public B {
private:
    int i;
public:
    D() { i = 1; }
    int g() { return i; }
    int h() { return i + 5; }
};
```

```
D d;
cout << d.g() << endl;
cout << d.f() << endl;
cout << d.h() << endl;
cout << d.B::h() << endl;
```

Inheritance

```
class B {
private:
    int i;
public:
    B() { i = 0; }
    int f() { return i; }
    int h() { return i + 10; }
};


class D : public B {
private:
    int i;
public:
    D() { i = 1; }
    int g() { return i; }
    int h() { return i + 5; }
};
```

```
D d;
cout << d.g() << endl;
cout << d.f() << endl;
cout << d.h() << endl;
cout << d.B::h() << endl;
```

h() will return 6 from the inherited class

h() will return 10 from the base class

Inheritance

```
class BB {
private:
    int i;
protected:
    int j;
public:
    BB() { i = 0; j = 100; }
    int f() { return i; }
    int h() { return i + 10; }
    int ff() { return j; }
    int hh() { return j + 100; }
    void x() { j = 2002; }
};
```

```
class DD : public BB {
private:
    int i;
public:
    DD() { i = 1; }
    int g() { return i; }
    int h() { return i + 5; }
    void x() { j = 1001; }
};
    DD d;
    cout << d.ff() << endl;
    cout << d.hh() << endl;
    d.x();
    cout << d.ff() << endl;
    cout << d.hh() << endl;
    d.BB::x();
    cout << d.ff() << endl;
    cout << d.hh() << endl;
```

Inheritance

```
DD d;
cout << d.ff() << endl;          100
cout << d.hh() << endl;          200
```

Inheritance

```
class BB {
private:
    int i;
protected:
    int j;
public:
    BB() { i = 0; j = 100; }
    int f() { return i; }
    int h() { return i + 10; }
    int ff() { return j; }
    int hh() { return j + 100; }
    void x() { j = 2002; }
};
```

```
class DD : public BB {
private:
    int i;
public:
    DD() { i = 1; }
    int g() { return i; }
    int h() { return i + 5; }
    void x() { j = 1001; }
};
    DD d;
    cout << d.ff() << endl;
    cout << d.hh() << endl;
    d.x();
    cout << d.ff() << endl;
    cout << d.hh() << endl;
    d.BB::x();
    cout << d.ff() << endl;
    cout << d.hh() << endl;
```

Inheritance

```
DD d;
cout << d.ff() << endl;          100
cout << d.hh() << endl;          200
d.x();
cout << d.ff() << endl;          1001
cout << d.hh() << endl;          1101
```

Inheritance

```
class BB {
private:
    int i;
protected:
    int j;
public:
    BB() { i = 0; j = 100; }
    int f() { return i; }
    int h() { return i + 10; }
    int ff() { return j; }
    int hh() { return j + 100; }
    void x() { j = 2002; }
};
```

```
class DD : public BB {
private:
    int i;
public:
    DD() { i = 1; }
    int g() { return i; }
    int h() { return i + 5; }
    void x() { j = 1001; }
};
    DD d;
    cout << d.ff() << endl;
    cout << d.hh() << endl;
    d.x();
    cout << d.ff() << endl;
    cout << d.hh() << endl;
    d.BB::x();
    cout << d.ff() << endl;
    cout << d.hh() << endl;
```

Inheritance

```
DD d;
cout << d.ff() << endl;
cout << d.hh() << endl;
d.x();
cout << d.ff() << endl;
cout << d.hh() << endl;
d.BB::x();
cout << d.ff() << endl;
cout << d.hh() << endl;
```

```
100
200

1001
1101

2002
2102
```

Inheritance

```
class BB {
private:
    int i;
protected:
    int j;
public:
    BB() { i = 0; j = 100; }
    int f() { return i; }
    int h() { return i + 10; }
    int ff() { return j; }
    int hh() { return j + 100; }
    void x() { j = 2002; }
};
```

```
class DD : public BB {
private:
    int i;
public:
    DD() { i = 1; }
    DD() { i = 0; j = 300; }
    int g() { return i; }
    int h() { return i + 5; }
    void x() { j = 1001; }
};
    DD d;
    cout << d.g() << endl;
    cout << d.f() << endl;
    cout << d.h() << endl;
    cout << d.BB::h() << endl;
    cout << d.ff() << endl;
    cout << d.hh() << endl;
```

Inheritance

```
class BB {
private:
    int i;
protected:
    int j;
public:
    BB() { i = 0; j = 100; }
    int f() { return i; }
    int h() { return i + 10; }
    int ff() { return j; }
    int hh() { return j + 100; }
    void x() { j = 2002; }
};
```

```
class DD : public BB {
private:
    int i;
public:
    DD() { i = 1; }
    DD() { i = 0; j = 300; }
    int g() { return i; }
    int h() { return i + 5; }
    void x() { j = 1001; }
};
    DD d;
    cout << d.g() << endl;
    cout << d.f() << endl;
    cout << d.h() << endl;
    cout << d.BB::h() << endl;
    cout << d.ff() << endl;
    cout << d.hh() << endl;
```

j will be 300 because BB's constructor will be called first

Questions?

Initialization in Composition and Inheritance

- Constructors are needed to initialize sub-objects and inherited class
- Default constructors are used for these sub-objects
- What if you want to initialize them with specific values for members (particularly if they are private)?
  - Call the specific constructor using the initializer list

Inheritance Constructor Initializer

```
1.  class Foo {
2.  private:
3.      int i;
4.  public:
5.      Foo() { i = 0; };
6.      Foo(int j) { i = j; };
7.      int read() { return i; }
8.  };
9.
10. class Bar : public Foo {
11. private:
12.     int i;
13. public:
14.     Bar() : Foo() { i = 0; }
15.     Bar(int j) : Foo(j + 1) { i = j; }
16.     void print() { cout << i << " " << read()
17.                    << endl;}
18. };
```

```
19.     Bar b1;
20.     Bar b2(10);
21.     b1.print();
22.     b2.print();
```

Composition Constructor Initializer

```
1.  class Foo {
2.  private:
3.      int i;
4.  public:
5.      Foo() { i = 0; };
6.      Foo(int j) { i = j; };
7.      int read() { return i; }
8.  };
9.  class Meh {
10. private:
11.     int i;
12.     Foo f;
13. public:
14.     Meh() : f() { i = 0; }
15.     Meh(int j) : f(j + 1) { i = j; }
16.     void print() { cout << i << " " << f.read()
17.                    << endl;}
18. };
```

```
19.     Meh m1;
20.     Meh m2(20);
21.     m1.print();
22.     m2.print();
```

Combine Inheritance & Composition

- You can combine them and they behave as described

Inheritance

- If you provide a new definition for an inherited function
  - If you provide the exact signature and return type – **redefining** for regular member functions and **overriding** when the base class member function is virtual (polymorphism)
  - What if you change the argument list and/or return type?

Name Hiding

```
class Base {
public:
  int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  int f(string) const { return 1; }
  void g() {}
};
class Derived1 : public Base {
public:
  void g()const {}
};
```

- Derived1 **redefines** g()
  - g() will refer to the redefined version for the inherited class
  - Base::g() is still accessible if called explicitly
  - Both overloaded version of f() are still available for use (because they are public)

Name Hiding

```
class Base {
public:
  int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  int f(string) const { return 1; }
  void g() {}
};
class Derived2 : public Base {
public:
  // Redefinition:
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
};
```

- Derived2 **redefines** the first of the overloaded versions of f() (but not the other)
  - Second overloaded f() is now also **not** available
    - Unless called explicitly by using `Base::f("hello");`
  - g() is available

Name Hiding

```
class Base {
public:
  int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  int f(string) const { return 1; }
  void g() {}
};
class Derived3 : public Base {
public:
  // Change return type:
  void f() const { cout << "Derived3::f()\n"; }
};
```

- Derived3 changes the return type of one of the overloaded f()
  - The other f() is **not** available
  - You can also say the first f() - without any arguments – is also not available (since it's a "different" function)
  - Both are still available by directly referring them (i.e., Base::f())
  - g() is available

Name Hiding

```
class Base {
public:
  int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  int f(string) const { return 1; }
  void g() {}
};
class Derived4 : public Base {
public:
  // Change argument list:
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};
```

- Derived4 changes the argument list
  - Also hides both overloaded base functions f()
  - But again, referred via Base::f()
  - g() is still available

Name Hiding

- Anytime you **redefine** an overloaded function name from the base class, or **change the argument list** or **return type**, all other versions are made unavailable to the inherited class
- If you do this, you are using the class in a different way than inheritance is normally intended to support – ultimate goal of inheritance it to support polymorphism – so it is considered as **changing the interface** of the base class, therefore **hiding all corresponding functions**

Static Member Functions

- Inherited to derived class
- If you redefine a static function, all other overloaded functions in the base class are hidden (as previously described)
- If you change the signature, all base class versions with the same name are hidden (as previously described)
- static member functions **cannot be virtual**

Questions?

Upcasting

- A derived class is **a type** of the base class

```
enum note {C, Csharp, Cflat};
class Instrument {
public:
    void play(note) const {}
};
class Wind : public Instrument {};
void tune(Instrument& i) {
    i.play(C);
}
void testSeven()
{
    Wind flute;
    tune(flute);
}
```

Upcasting

```
enum note {C, Csharp, Cflat};
class Instrument {
public:
    void play(note) const {cout << "Instrument" << endl; }

};
class Wind : public Instrument {
public:
    void play(note) const {cout << "Wind" << endl; }
};
void tune(Instrument& i) {
    i.play(C);
}
```

```
int main()
{
    Wind w;
    Instrument* ip = &w;  // upcast
    Instrument& ir = w;   // upcast
    tune(*ip);
    return 0;
}
```

Upcasting

```
enum note {C, Csharp, Cflat};
class Instrument {
public:
    virtual void  play(note) const { cout << "Instrument" << endl; }

};
class Wind : public Instrument {
public:
    void play(note) const { cout << "Wind" << endl; }
};
void tune(Instrument &i) {
    i.play(C);
}
```

```
int main()
{
    Wind w;
    Instrument *ip = &w;  // upcast
    Instrument &ir = w;  // upcast
    tune(*ip);
    return 0;
}
```

Polymorphism is enabled through late-binding.
Binding is the process by which a compiler converts identifiers to machine address.
Through late-binding, the runtime (as opposed to the compiler) identifies that the address is that of the derived class, and knows to use the polymorphed function

Questions

Upcasting and Copy-Constructor

```
class Parent {
private:
    int i;
public:
    Parent(int ii) : i(ii) { cout << "Parent(int ii)" << endl; }
    Parent(const Parent& b) : i(b.i) {
        cout << "Parent(const Parent &b)" << endl;
    }
    Parent() : i(0) { cout << "Parent()" << endl; }
    friend ostream& operator<< (ostream &os, const Parent &b) {
        return os << "Parent: " << b.i << endl;
    }
};
    Parent p(101);
    cout << p;
```

Upcasting and Copy-Constructor

- `Parent(int ii)`
- `Parent: 101`

Upcasting and Copy-Constructor

```
class Member {
private:
    int i;
public:
    Member(int ii) : i(ii) { cout << "Member(int ii)" << endl; }
    Member(const Member &m) : i(m.i) {
        cout << "Member(const Member &m)" << endl;
    }
    friend ostream& operator<< (ostream &os, const Member &m) {
        return os << "Member: " << m.i << endl;
    }
};
    Member m(202);
    cout << m;
```

Upcasting and Copy-Constructor

- `Member(int ii)`
- `Member: 202`

Upcasting and Copy-Constructor

```
class Child : public Parent {
private:
    int i;
    Member m;
public:
    Child(int ii) : Parent(ii), i(ii), m(ii) {
        cout << "Child(int ii)" << endl;
    }
    friend ostream& operator<< (ostream& os, const Child& c) {
        return os << (Parent&) c << c.m << "Child: " << c.i << endl;
    }
};
    Child c(303);
    cout << c;
```

Upcasting and Copy-Constructor

- `Parent(int ii)`
- `Member(int ii)`
- `Child(int ii)`
- **`Parent: 303`**
- `Member: 303`
- `Child: 303`

From the constructor call
Child c(303)

```
os << (Parent&) c << c.m << "Child: " << c.i << endl;
```

Upcasting and Copy-Constructor

- `Parent(int ii)`
- `Member(int ii)`
- `Child(int ii)`
- `Parent: 303`
- **`Member: 303`**
- `Child: 303`

From the constructor call
Child c(303)

```
os << (Parent&) c << c.m << "Child: " << c.i << endl;
```

Upcasting and Copy-Constructor

- `Parent(int ii)`
- `Member(int ii)`
- `Child(int ii)`

From the constructor call
Child c(303)

- `Parent: 303`
- `Member: 303`
- **`Child: 303`**

```
os << (Parent&) c << c.m << "Child: " << c.i << endl;
```

Questions?

Upcasting and Copy-Constructor

- `Child c2(2);`
- `Child c3 = c2; // Remember that Child class has no copy-constructor`
- `cout << c3;`

Upcasting and Copy-Constructor

- `Parent(int ii)`
- `Member(int ii)`
- `Child(int ii)`
- **`Parent(const Parent &b)`**
- **`Member(const Member &m)`**
- `Parent: 2`
- `Member: 2`
- `Child: 2`

Since Child has no explicitly defined copy-constructor, the compiler will **synthesize** one by calling the **Parent copy-constructor** and the **Member copy-constructor**

Upcasting and Copy-Constructor

-

Upcasting and Copy-Constructor

- What if you create a copy-constructor but make a mistake and leave out the base class' constructor
  - `Child(const Child& c) :` ~~`Parent(c.i)`~~`, i(c.i), m(c.m) {}`
- What would happen?
  - The default constructor will automatically be called for the base-class part of the Child (i.e., Parent), since that's what the compiler falls back on when it has no other choice of constructor to call

Upcasting and Copy-Constructor

```
class Parent {
private:
    int i;
public:
    Parent(int ii) : i(ii) { cout << "Parent(int ii)" << endl; }
    Parent(const Parent &b) : i(b.i) {
        cout << "Parent(const Parent &b)" << endl;
    }
    Parent() : i(0) { cout << "Parent()" << endl; }
    friend ostream& operator<< (ostream &os, const Parent &b) {
        return os << "Parent: " << b.i << endl;
    }
};
```

Upcasting and Copy-Constructor

- `Parent(int ii)`
- `Member(int ii)`
- `Child(int ii)`
- `Parent(const Parent &b)`
- `Member(const Member &m)`
- `Parent: 2`
- `Member: 2`
- `Child: 2`

Upcasting and Copy-Constructor

- `Parent(int ii)`
- `Member(int ii)`
- `Child(int ii)`
- `Parent()`
- `Member(const Member &m)`
- `Parent: 0`
- `Member: 2`
- `Child: 2`

Composition vs. Inheritance

- Depends on the application at hand
- General rule of thumb – composition is generally used when you want the **features of an existing class inside your class, but not its interface** (i.e., the user sees the interface you've defined rather than the original class)