COMP6771
Advanced C++ Programming

Week 11
Object Oriented Programming

2016

www.cse.unsw.edu.au/~cs6771

Object-Oriented Programming in C++

1. Inheritance: ability to create new classes based on existing ones — *supported by class derivation*
2. Polymorphism: allows objects of a class to be used as if they were objects of another class
3. Dynamic binding: run-time resolution of the appropriate function to invoke based on the type of the object.
   2 & 3 are supported via virtual functions in C++

Thinking about Programming in C++

- Represent the concepts in your problem using classes
- Represent their relations using inheritance
- Untangle the dependency between the classes by separating the interface of a class from its implementation

Problem: Representing the Kinds of Books in a Bookstore

- Identify concepts: books, discounted books, used books, ...
- Choose classes to represent the concepts:

`Book`,	`Disc_book`,	`Used_book`, ...
`(undiscounted books)`	`(discounted)`	`(used)`

- Identify the relationships using inheritance:

```
class Book { ... }
class Disc_book: public Book { ... }
class Used_book: public Book { ... }
```

- public is the same as Java's extends
- Class hierarchy:

Defining a Base Class (for Undiscounted Books)

```
1  class Book {
2  public:
3    Book (const std::string &book_isbn = "",
4                double sales_price = 0.0)
5                : isbn_no{book_isbn}, price{sales_price} { }
6
7    std::string isbn() const { return isbn_no; }
8
9    virtual double net_price(std::size_t n) const
10               { return n * price; }
11
12   virtual ~Book() { }
13
14 private:
15     std::string isbn_no;
16
17 protected:
18     double price;
19 };
```

Defining a Base Class (for Undiscounted Books)

- The interface of Book: all its public members
- Private and protected members inaccessible in user code
- Two kinds of functions:
  - virtuals:
    - Preceded by the keyword virtual
    - Dynamic binding turned on
    - Expected to be overridden by a derived class
  - nonvirtuals:
    - Static binding
    - Expected to be inherited by a derived class
- Virtual destructors (next week)

Defining a Derived Class (for discounted Books)

```
1  class Disc_book : public Book {
2  public:
3    virtual double net_price(std::size_t n) const;
4
5  private:
6     std::size_t min_qty;
7     double discount;
8  };
9
10 double Disc_book::net_price(std::size_t n) const {
11   if (n >= min_qty)
12     return n * (1 - discount) * price;
13   else
14     return n * price;
15 }
```

optional, but should use

Public Inheritance

- Almost universally used in practice
- A derived class inherits all public and protected members as if they were declared in the derived class:
  - Public members remain public
  - Protected members remain protected
  - **Private members are inaccessible**
- Known as the interface inheritance
- C++'s equivalent of Java's class inheritance (by extend)
- Name hiding causes exceptions to the rule — BAD (next week)
- Private/protected inheritance (with ": public" replaced by ": private/protected") may be discussed later
- Friendship is not inherited

Member Access Control

```
class Foo {
public:
  Members accessible by everyone
protected:
  Members that behave as public members
  to the derived classes and private members
  to the rest of the program
private:
  Accessible only by members & friends
}
```

- C++ classes support:
  - information hiding
  - encapsulation
- Friends are part of the public interface

C++ Protected Access

- A derived object may access the protected members of its base class only through a derived object
- Does this compile?

```
1  class ABC_Phone {
2  protected:
3    void ring() { };
4    int secret;
5  };
6
7  class Phone_Maker_1 : public ABC_Phone { };
8
9  class Phone_Maker_2 : public ABC_Phone {
10   void f(Phone_Maker_1 *p) {
11      p->ring();
12      std::cout << p->secret;
13   }
14 };
```

Inheritance Relationship in Public Inheritance

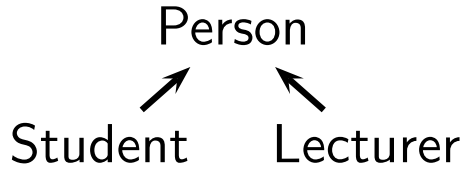
- The relation between Book and Disc_book

```
Disc_book "a kind of a"       Book
          "derived from"      Book
          "a specialised"     Book
          "subclass" of       Book
          "derived class" of Book

Book is   the "base class" of Disc_book
          the "superclass" of Disc_book
```

- Can use a Disc_book wherever a Book is expected, but not vice versa

Inheritance Relationship in Public Inheritance




```
class Person { };
class Student : public Person { };
class Lecturer : public Person { };
```

- The relation between Student and Person

```
Student is "a kind of a"       Person
           "derived from"      Person
           "a specialised"     Person
           "subclass" of       Person
           "derived class" of Person

Person is  the "base class" of Student
           the "superclass" of Student
```

- Can use a Student wherever a Person is expected, but not vice versa

Java vs. C++

- In Java, all methods (i.e., functions) are virtual
- In C++, there are two possibilities:
  - all functions are defaulted to be nonvirtual
  - functions are made virtual using the keyword

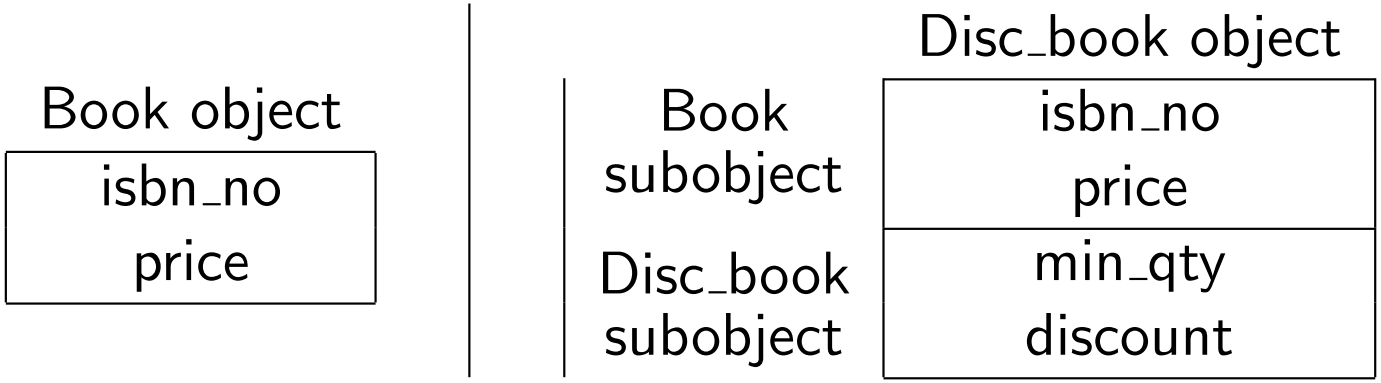
virtual

Final

- Originally distinguished for efficiency considerations:
- The final specifier introduced in C++11, as in Java:

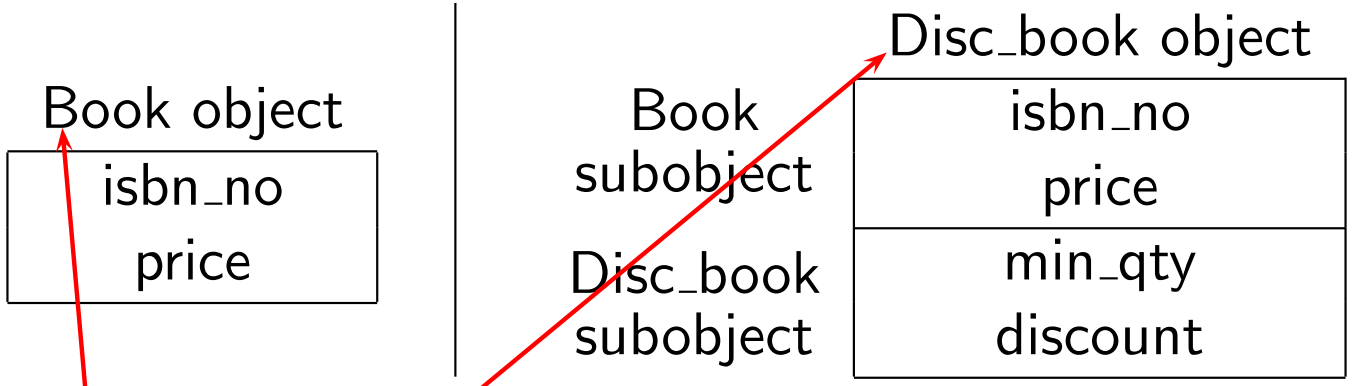
```
1  #include<iostream>
2  class Base {
3    virtual void f(int) { }
4  };
5
6  class Derived : public Base {
7    void f(int) final { }
8  };
9
10 class MostDerived : public Drived {
11   void f(int) { } // error: cannot be overridden
12 };
```

Memory Layout for Data Members in Objects



- Only nonstatic members are stored in objects
- Disc_book is more specialised and thus contains more info

A Disc_book is-a Book But Not Vice Versa

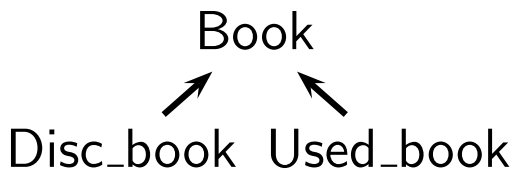


- Up-casting (derived-to-base) conversions:

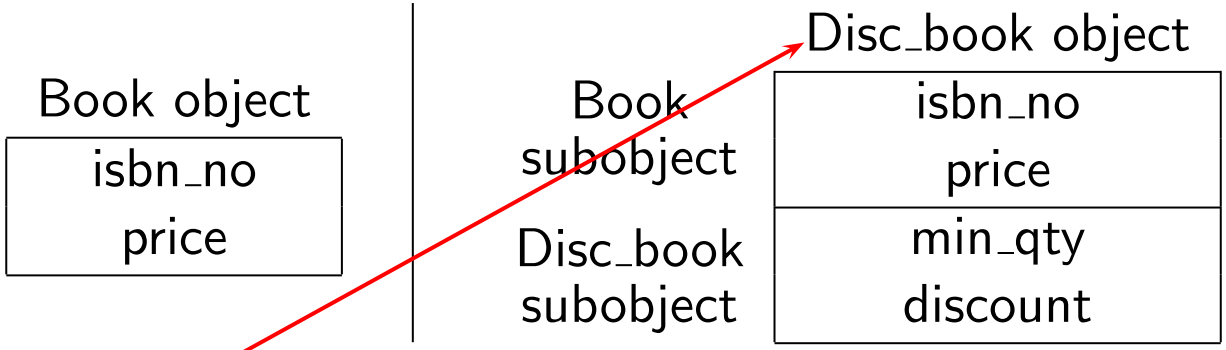
```
Disc_book db;
Book *p = &db; // ok
Book &p = db;  // ok
```

- Down-casting (base-to-derived) conversions:

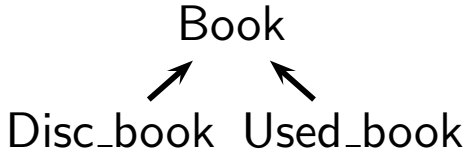
```
Book b;
Disc_book *p = &b; // error
Disc_book &p = b;  // error
```



A Disc_book is-a Book But Not Vice Versa



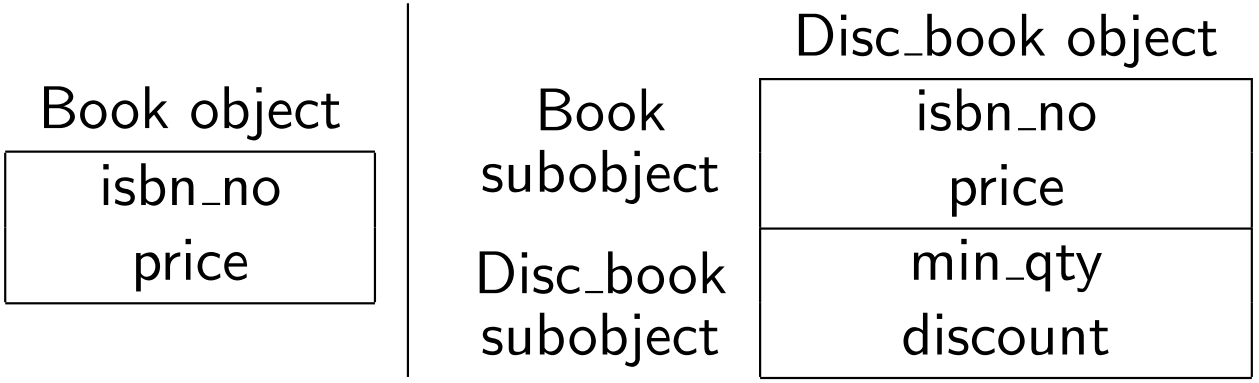

```
Disc_book db;
Book *p = &db; // ok (up-casting)
Disc_book *q;
```




```
q = p;          // error (down-casting)
q = static_cast<Disc_book*> p; // ok, *p is a Disc_book

try {
  q = dynamic_cast<Disc_book*> p; // ok (if unsure)
} catch(...)   {  ... }           // dynamically checked
```

A Disc_book is-a Book But Not Vice Versa



- Object Slicing:

```
Disc_book db;
Book b = db; // only the base part of db copied
```

- Container and inheritance don't mix

```
multiset<Book> basket;
Book b;
Disc_book db
basket.insert(b);  // add a copy of b
basket.insert(db); // add a copy of db sliced
                   // down to the base part only
```

Some User Code

```
1  void print_total(std::ostream &os,
2                   const Book &b, size_t n) {
3
4    os << "ISBN: " << b.isbn()  // always call Book:isbn()
5       << " number sold: " << n << " total price: "
6       << b.net_price(n) << std::endl;
7  };
8
9  int main() {
10   Book b{"Book 1",9.99};
11   Disc_book db; // no inherited constructor
12   print_total(std::cout, b, 10);  // call Book::net_price()
13   print_total(std::cout, db, 10); // call Disc_book::net_price()
14 }
```

Static Type and Dynamic Type of Class Objects

- The static or declared type at the declaration
- The dynamic type of the object pointed or referred to

declaration	object pointed/referenced by p: static	dynamic
`Disc_book db;` `Book *p = &db;`	Book	Disc_book
`Disc_book db;` `Book &p = db;`	Book	Disc_book

Static and Dynamic Typing/Binding

- C++:
  - Statically typed
  - Static binding for nonvirtuals (based on static type of receiver)
  - Dynamic binding for virtuals (based on dynamic type of receiver)
- Java:
  - Statically typed
  - Dynamic binding only (all functions are virtual)
- Dynamically typed languages: Smalltalk and APL
- For the pros and cons of static and dynamic typing, see:

  `Robert W. Sebesta, Concepts of Programming Languages, 10th Ed, Addison-Wesley.`

Static: compile-time

Dynamic: run-time

Static Binding for Nonvirtuals

```
Book b;
Disc_book db;
Book *p = &b, *q = &db;
```

- `p->isbn();`
  - Static typing: static type of p is Book
  - Static binding: call Book::isbn()
- `q->isbn();`
  - Static typing: static type of q is Book
  - Static binding: call Book::isbn()
- Similarly if p and q are references

  ```
  Book &p = b, &q = db;
  ```

Dynamic Binding for Virtuals

- Can be achieved with pointers to base classes:

```
Disc_book db;
Book *pb = &db;
Disc_book *pd = &db;
pb->net_price(); // call Disc_book::net_price()
pd->net_price(); // call Disc_book::net_price()
```

- Static typing:

```
pb->net_price(); // Book::net_price() exists
pd->net_price(); // Disc_book::net_price() exists
```

- Dynamic binding: runtime resolution of the function called based on the dynamic type of the object
- net_price is said to be polymorphic since we can also have:

```
Book b;
Book *p = &b;
p->net_price() // call Book::net_price()
```

- Can also be achieved via references to base classes

Data Types Revisited

- A type: a set of values and a set of operations on the values
- Book:
  - set of values: Book, Disc_book objects, ...
  - set of operations, e.g., Book::isbn()
    - The interface of Book is still the same as before
    - But its implementation can be changed in Disc_book
- Polymorphic type: a type with virtual functions
- Note: static typing works as before

Pure Virtual Functions

- A class is an abstract base class (ABC) if it contains some pure virtual functions (declared or inherited)

```
class shape {
public:
   virtual void draw() = 0;    // pure virtual
};

class circle : public shape {
   void draw() { /* draw a circle */ }
};

class square: public shape {
   void draw() { /* draw a square */ }
};
```

- ABC's are similar to Java's interfaces
- No objects can be constructed from ABC's

C++ Member Functions

Syntax	Name	Conceptual Meaning
virtual void draw()=0	pure virtual	inherit interface only
virtual void net_price()	virtual	inherit interface & an optional impl.
std::string isbn()	nonvirtual	inherit interface & a mandatory impl. invariance over specialisation — can be broken immorally with name hiding (next week)

Reading

- Chapter 15
- Chapter 12, Stroustrup's book (Derived Classes)
- C++ FAQs 20.09, 20.12

Next Lecture: C++ Object Model and Copy Control