ST. CHARLES COMMUNITY COLLEGE

CPT-182 - Programming in C++

Module 9

Class Inheritance, Polymorphism

Dayu Wang

- **Derived Classes**

→ **Commonly, one class is similar to another class but with some additions or variations.**

→ **[Example]** `Generic_Item` **class**

A store inventory system might use a class called `Generic_Item` **having** `name` **and** `quantity` **members.**

In "Generic_Item.h"

```
class Generic_Item {
public:
    // Constructor with initial values of "name" and "quantity".
    Generic_Item(const string& = "", unsigned int = 0);
    // Getters and setters
    string get_name() const;
    unsigned int get_quantity() const;
    void set_name(const string&);
    void set_quantity(unsigned int);
    // Class-member function
    virtual void print(ostream&) const;
private:
    // Data fields
    string name;  // Stores the name of the item.
    unsigned int quantity;  // Stores the quantity of the item.
};
```

In "Generic_Item.cpp"

```
// Constructor with initial values of "name" and "quantity".
Generic_Item::Generic_Item(const string& name, unsigned int quantity) :
    name(name), quantity(quantity) {}

// Getters
string Generic_Item::get_name() const { return name; }
unsigned int Generic_Item::get_quantity() const { return quantity; }

// Setters
void Generic_Item::set_name(const string& name) { this->name = name; }
void Generic_Item::set_quantity(unsigned int quantity) {
    this->quantity = quantity;
}

// Class-member function

/** Writes the item to an output stream.
    @param out: an output stream to write the item
*/
void Generic_Item::print(ostream& out) const {
    out << "Name: " << name << endl;
    out << "Quantity: " << quantity << endl;
}
```


But for produce (fruits and vegetables), a class `Produce_Item` having `name`, `quantity`, and `expiration_date` members may be desired.

[Fact] Produce is a generic item.

Just like square is a shape.

Generic_Item	Produce_Item
`string name`	`string name`
`unsigned int quantity`	`unsigned int quantity`
	`string expiration_date`
`Generic_Item(const string& = "", unsigned int = 0)` **Variation**	`Produce_Item(const string& = "", unsigned int = 0, const string& = "")`
`string get_name() const`	`string get_name() const`
`unsigned int get_quantity() const`	`unsigned int get_quantity() const`
`void set_name(const string&)`	`void set_name(const string&)`
`void set_quantity(unsigned int)`	`void set_quantity(unsigned int)`
Addition	`string get_expiration_date() const`
	`void set_expiration_date(const string&)`
`void print(ostream&) const`	`void print(ostream&) const`

Produce_Item is Generic_Item with <u>additional features</u> or <u>varied features</u>.

Ideally a program could define the Produce_Item class as being the same as the Generic_Item class.

Only <u>define the difference</u> with Generic_Item in Produce_Item.

→ Such similarity among classes is supported by indicating that <u>a class is derived from another class</u>.

→ [Example] The Produce_Item class

How to indicate that Produce_Item is derived from Generic_Item?

```
class Produce_Item : public Generic_Item {
    // Class definition
}
```

In C++, sometimes derived class is also called child class or subclass.

In this example, Generic_Item is called the base class.

In this example, Produce_Item is called the derived class.

What happens after "public Generic_Item"?

1) All the private and public attributes defined in the Generic_Item class will be "<u>imported</u>" into the Produce_Item class.

   This is called inheritance.

2) All the public attributes in Generic_Item class become public attributes of Produce_Item class.

<u>Data fields</u> in class Produce_Item

```
class Produce_Item : public Generic_Item {
private:
    // Data fields
    string expiration_date;

    // Do you need to define name and quantity in Produce_Item?
}
```

We only need to add <u>additional variables</u> to the derived class.

<u>Constructors</u> in class Produce_Item

```
Produce_Item(const string& = "", unsigned int = 0, const string& = "");
```

```
Produce_Item::Produce_Item(const string& name, unsigned int quantity,
                           const string& expiration_date) {
    set_name(name);
    set_quantity(quantity);
    this->expiration_date = expiration_date;
}
```

→ **Why we must use `set_name()` and `set_quantity()` to initialize `name` and `quantity`?**

`name` and `quantity` are `private` attributes of the base class.

They are not ~~accessible~~ in the derived class.

```
Produce_Item::Produce_Item(const string& name, unsigned int quantity,
                           const string& expiration_date) {
    this->name = name;
    this->quantity = quantity;
    this->expiration_date = expiration_date;
}
```

```
Console  Member Generic_Item::name is inaccessible.
```

`set_name()` and `set_quantity()` are `public` attributes of the base class.

So, they are accessible in the derived class.

```
Produce_Item::Produce_Item(const string& name, unsigned int quantity,
                           const string& expiration_date) {
    set_name(name);   set_quantity(quantity);
    this->expiration_date = expiration_date;
}
```

`expiration_date` is also `private`, but why it is accessible?

It is declared in the derived class (not the ~~base class~~).

`private` attributes are accessible within the same class.

→ **The base class's constructor initializes data fields `name` and `quantity`.**

- The constructor is `public` in the base class.
- The constructor should be accessible in the derived class (`Produce_Item`).
- Can we use the base class's constructor to initialize `name` and `quantity` in a derived class?

In "Produce_Item.cpp"

```
Produce_Item::Produce_Item(const string& name, unsigned int quantity,
    const string& expiration_date) : Generic_Item(name, quantity),
    expiration_date(expiration_date) {}
```

→ **Access variables**

- `private`

  `private` attributes are accessible only within the same class.

  Even if for a derived class, it is not ~~the same class~~.

  `private` attributes are not accessible in derived classes.

- `public`

  `public` attributes are accessible outside the class.

  They are accessible everywhere in the source code of the same project.

- `protected`

  `protected` **attributes are accessible within the same class, and are also <u>accessible in derived classes</u>.**

  `protected` **attributes are not accessible in other classes.**

```
class Generic_Item {
protected:
    // Data fields
    string name;
    unsigned int quantity;
    // ...
}
```

```
Produce_Item::Produce_Item(const string& name, unsigned int quantity,
                           const string& expiration_date) {
    this->name = name;
    this->quantity = quantity;
    this->expiration_date = expiration_date;
}
```

Correct

→ **<u>Getters and setters</u> in class** `Produce_Item`

```
string Produce_Item::get_expiration_date() const {
    return expiration_date;
}

void Produce_Item::set_expiration_date(const string& expiration_date) {
    this->expiration_date = expiration_date;
}
```

→ **Other facts**

- **Any class may serve as a base class.**

  No changes to the definition of that class are required.
- **The derived class is said to inherit the properties of its base class, a concept commonly called inheritance.**
- **An object declared of a derived class type has access to all the** `private` **and** `public` **members of the derived class as well as the** `public` **members of the base class.**

- **<u>Overriding member functions</u>**

  → **We start with an experiment.**

```
int main() {
    Generic_Item item_1("Hat", 10);
    Produce_Item item_2("Egg", 5, "04/20/2023");

    item_1.print(cout);
    cout << endl;
    item_2.print(cout);

    system("pause");
    return 0;
}
```

Console

```
Name: Hat
Quantity: 10

Name: Egg
Quantity: 5
```

The expiration date of `item_2` (`Produce_Item`) is not shown.

→ **`print()` is a `public` function defined in the base class (`Generic_Item`).**

It is also accessible in derived classes (e.g., `Produce_Item`).

For `Generic_Item`, since it only has `name` and `quantity`, output these two is fine.

However, for `Produce_Item`, it has `expiration_date` as well, so output only `name` and `quantity` is not good enough.

→ **What we want is the following:**

- **`print()` is accessible in both `Generic_Item` and `Produce_Item`.**
- **If the current object (e.g., `item_1`) is `Generic_Item`, then calling `item_1.print()` will output its `name` and `quantity`.**
- **If the current object (e.g., `item_2`) is `Produce_Item`, then calling `item_2.print()` will output its `name`, `quantity`, and `expiration_date`.**

→ **The solution is called overriding.**

- **A derived class may define a member function <u>having the same name and parameter types</u> as the base class.**
- **Such a member function overrides the function of the base class.**
- **In overriding, a derived class member function takes precedence over base class member function with the same name and parameters.**

→ **Overriding allows the same member function in the base class to have <u>different behavior</u> in derived classes.**

→ How to write overriding function?

```
class Generic_Item {
public:
    // Constructor
    Generic_Item(const string& = "", unsigned int = 0);
    // Getters
    string get_name() const;
    unsigned int get_quantity() const;
    // Setters
    void set_name(const string&);
    void set_quantity(unsigned int);
    // Class-member functions
    virtual void print(ostream&) const;
private:
    // Data fields
    string name;
    unsigned int quantity;
};
```

- **Add a `virtual` keyword in the base class function definition.**

  The `virtual` keyword indicates that the function is overridable in derived classes.


```
class Produce_Item : public Generic_Item {
public:
    // Constructor
    Produce_Item(const string& = "", unsigned int = 0, const string& = "");
    // Getter
    string get_expiration_date() const;
    // Setter
    void set_expiration_date(const string&);
    // Class-member function
    void print(ostream&) const;
private:
    // Data field
    string expiration_date;
};
```

- **The overriding function must have <u>same access variable</u>, <u>same return type</u>, <u>same function name</u>, and <u>same parameter list</u> in derived class.**

```
void Produce_Item::print(ostream& out) const {
    out << "Name: " << get_name() << endl;
    out << "Quantity: " << get_quantity() << endl;
    out << "Expiration date: " << expiration_date << endl;
}
```

- **Define the function in derived class.**

```
void Generic_Item::print(ostream& out) const {
    out << "Name: " << get_name() << endl;
    out << "Quantity: " << get_quantity() << endl;
}
```

```
void Produce_Item::print(ostream& out) const {
    out << "Name: " << get_name() << endl;
    out << "Quantity: " << get_quantity() << endl;
    out << "Expiration date: " << expiration_date << endl;
}
```

Can we call the base class version of the `print()` function in derived class?

```
void Produce_Item::print(ostream& out) const {
    Generic_Item::print(out);
    out << "Expiration date: " << expiration_date << endl;
}
```



```
int main() {
    // No change of the main() function.

    Generic_Item item_1("Hat", 10);
    Produce_Item item_2("Egg", 5, "04/20/2023");

    item_1.print(cout);
    cout << endl;
    item_2.print(cout);

    system("pause");
    return 0;
}
```

Console

```
Name: Hat
Quantity: 10

Name: Egg
Quantity: 5
Expiration date: 04/20/2023
```

- **Other Examples**

→ **A derived class can itself serve as a base class for another class.**

```
class Fruit_Produce : public Produce_Item {
public:
    Fruit_Produce(const string& = "", unsigned int = 0,
                  const string& = "", bool = false);
    bool get_has_seed() const;
    void set_has_seed(bool);
    void print(ostream&) const;
private:
    bool has_seed;
};
```

```
Fruit_Produce::Fruit_Produce(const string& name, unsigned int quantity,
    const string& expiration_date, bool has_seed) :
    Produce_Item(name, quantity, expiration_date), has_seed(has_seed) {}
bool Fruit_Produce::get_has_seed() const { return has_seed; }
void Fruit_Produce::set_has_seed(bool has_seed) {
    this->has_seed = has_seed;
}
void Fruit_Produce::print(ostream& out) const {
    Produce_Item::print(out);
    out << "Has seed: " << (has_seed ? "true" : "false") << endl;
}
```

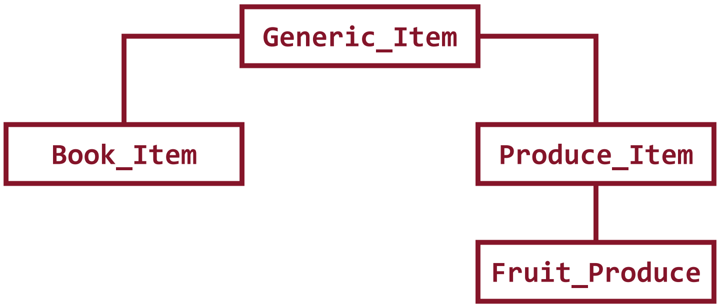

→ **A class can serve as a base class for multiple derived classes.**

```
class Book_Item : public Generic_Item {
public:
    // Constructor
    Book_Item(const string& = "", unsigned int = 0, const string& = "");
    // Getter
    string get_isbn() const;
    // Setter
    void set_isbn(const string&);
    // Class-member function
    void print(ostream&) const;
private:
    // Data field
    string isbn;
};
```

```
Book_Item::Book_Item(const string& name, unsigned int quantity,
    const string& isbn) : Generic_Item(name, quantity), isbn(isbn) {}
string Book_Item::get_isbn() const { return isbn; }
void Book_Item::set_isbn(const string& isbn) { this->isbn = isbn; }
void Book_Item::print(ostream& out) const {
    Generic_Item::print(out);
    out << "isbn: " << isbn << endl;
}
```

- **Polymorphism**

→ **The current classes (with their relationship) we have is as below.**



- **Create a pointer to the base class,** `Generic_Item`**.**

```
int main() {
    Generic_Item* item = NULL;
}
```

- **How can we instantiate the object** `item`**?**

```
int main() {
    Generic_Item* item = new Generic_Item("Hat", 10);
}
```

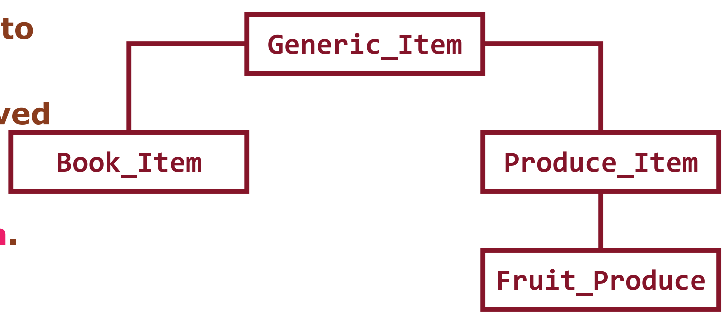
We can instantiate `item` **as a** `Generic_Item` **object.**


```
Generic_Item* item_1 = new Generic_Item("Hat", 10);
Generic_Item* item_2 = new Produce_Item("Egg", 5, "04/20/2023");
Generic_Item* item_3 = new Fruit_Produce("Apple", 10, "04/20/2023", true);
Generic_Item* item_4 = new Book_Item("Java History", 10, "103948593843");
```

- **You can use the base class to declare the pointer.**

  Then, you can use any derived class to instantiate the object.

  This is called polymorphism.



→ **Why this is important?**

- **What if the data type cannot be determined when you write the code?**

  For example, data type information is stored in the input file.

- **Polymorphism can determine the data type information at runtime.**

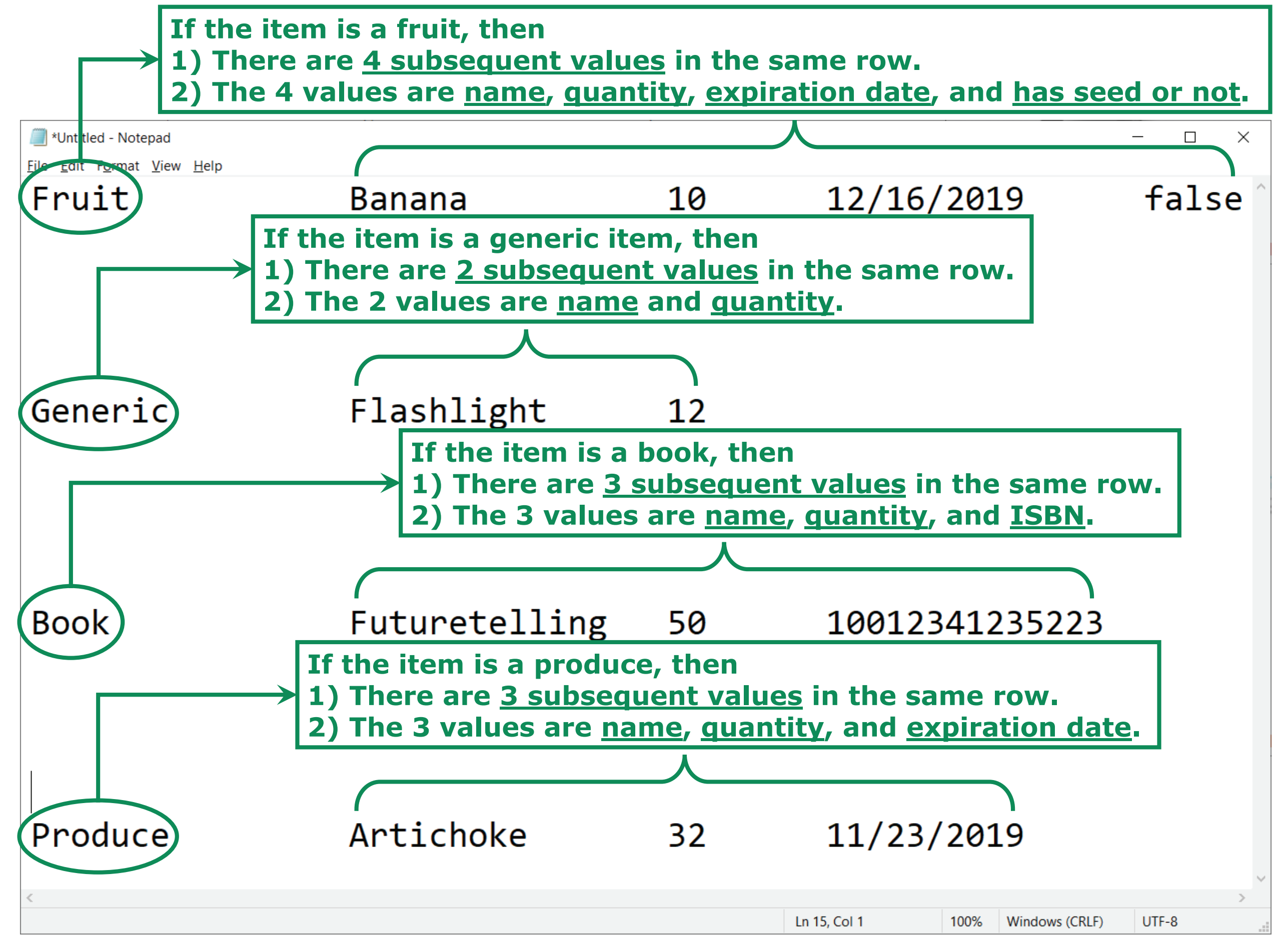
→ **[Example] Store inventory**

- **The input file lists all the items.**

  Each item is a separate line.

  The first value in each line is the type of the item.

  (e.g., "Generic", "Book", "Produce", or "Fruit")



→ **Store all the items in the input file to <u>a single vector</u>.**

```
int main() {
    vector<?> inventory;
}
```

What data type you need to put to replace '?'?

→ **Finally, output the all the inventory items stored in the vector to the output file.**

You need to call the `print()` function for each item in the vector.

- **<u>What are the differences between overloading and overriding?</u>**

→ **In overloading, functions with the <u>same name</u> must have <u>different parameter types</u>.**

→ **In overriding, a derived class member function takes precedence over base class member function with the <u>same name and parameter types</u>.**

→ **This is a popular job interview question.**

Only understanding overloading and overriding at this level cannot let you get a job.

→ **Actually, the function name and/or parameter types are just the appearance.**

We need to discuss the essence of overloading and overriding.

- **How can the compiler know which "version" to use?**

→ **Compiler must compile the code of the program before running the program.**

→ **Overloading**

```
void reverse(string& s) {
    int i = 0, j = s.size() - 1;
    while (i <= j) { swap(s[i++], s[j--]); }
}
```

```
void reverse(vector<int>& vec) {
    int i = 0, j = vec.size() - 1;
    while (i <= j) { swap(vec.at(i++), vec.at(j--)); }
}
```

Compiler can see the function argument.

1) If it is a string, then use the top version.

2) If it is a vector of integers, then use the bottom version.

Code can be compiled correctly without ~~running the program~~.

→ **Overriding**

```
item->print();
```

Does compiler use the "generic version", "produce version", "fruit version", or "book version" to compile the code?

It depends on the data type of `item`:

1) If `item` is `Generic_Item`, then use the "generic version".

2) If `item` is `Produce_Item`, then use the "produce version".

3) If `item` is `Fruit_Produce`, then use the "fruit version".

4) If `item` is `Book_Item`, then use the "book version".

However, compiler does not know what data type `item` is.

- **As discussed before, sometimes the data type of `item` depends on the input file.**
- **Compilation comes before execution.**

→ **Now, which "version" compiler will use to compile the code?**

- **Actually, compiler does not know which "version" to use.**
- **Therefore, this piece of code cannot be ~~compiled at the time of program compilation~~.**
- **At program compilation time, compiler will put a "mark" at this line of code.**

Later, at program runtime, when program execution reaches this line...

1) Program execution will be paused.
2) Complier will be awaken again.
3) Since at runtime, which object is calling the function is clear, compiler now can compile the code.
4) After the runtime compilation, program will continue to run.

→ What is the essence of overloading *versus* overriding?

They are compiled at different times.

Overloading is compiled at program compilation time.

Overriding is compiled at program runtime.

- **Abstract Classes**

→ An abstract class is a class that guides the design of derived classes but cannot ~~itself be instantiated as an object~~.

The philosophy is similar to how human beings abstract the nature.

For example, is "mammal" an actual animal?

→ An abstract class is a class that cannot be ~~instantiated as an object~~, but is the base class for some derived classes and specifies how the derived classes must be implemented.

A concrete class is a class that is not ~~abstract~~, and hence can be instantiated.

- **[Example] The `Shape` Class**

→ How to indicate that the class is an abstract class?

```
class Shape {
    // Nothing changed in the header.
};
```

→ Grab a pen and a piece of paper, write down what kinds of attributes a shape has?

[Important] Which functions you know how to implement and which functions you do not know how to implement?

Attribute Name	Can implement?
`double x`	
`double y`	
`double get_x() const` **and** `double get_y() const`	Yes
`void set_x(double)` **and** `void set_y(double)`	Yes
`double area() const`	No
`double perimeter() const`	No

→ **Why there are no ~~constructors~~?**

- **Constructors are used to instantiate a class.**
- `Shape` **is an abstract class which cannot ~~be instantiated~~.**
- **So, there are no ~~constructors~~ in the** `Shape` **class.**

→ **Why we cannot implement function** `area()`**?**

- **Different shapes use difference formulas to calculate the area.**
- **Square: area = side_length^2**
- **Circle: area = pi * radius^2**
- **Without the information of what kind of shape it is, we cannot ~~implement the `area()` function~~.**

→ **Due to the same reason, we cannot ~~implement the `perimeter()` function~~.**

→ **In the** `Shape` **class, functions** `area()` **and** `perimeter()` **are set to pure virtual functions.**

```
class Shape {
public:
    // Getters
    double get_x() const;
    double get_y() const;
    // Setters
    void set_x(double);
    void set_y(double);
    // Functions
    virtual double area() const = 0;
    virtual double perimeter() const = 0;
private:
    // Data fields
    double x, y;
};
```

- **Add** `virtual` **keyword at the beginning and add "**`= 0`**" at the end, then the function becomes a pure virtual function.**
- **Pure virtual functions are also called abstraction functions.**
- **A class may have many member functions. But if any one function is pure virtual, then the entire class is abstract (cannot ~~be instantiated~~).**

→ **The reason why abstract class cannot be implemented is that the class has at least one function that cannot be implemented (pure virtual).**

If all the functions can be implemented, then the class can be instantiated. Then, why make the class abstract?

- **Writing derived classes of an abstract class**

→ **[Example] The `Square` class.**

A derived class of an abstract class must override all the pure virtual functions in the base class.

```
class Square : public Shape {
public:
    // Constructor
    Square(double = 0, double = 0, double = 0);
    // Getter
    double get_side() const;
    // Setter
    void set_side(double);
    // Class-member functions
    double area() const;
    double perimeter() const;
private:
    // Data field
    double side;
};
```