CS601: Software Development for Scientific Computing

Autumn 2023

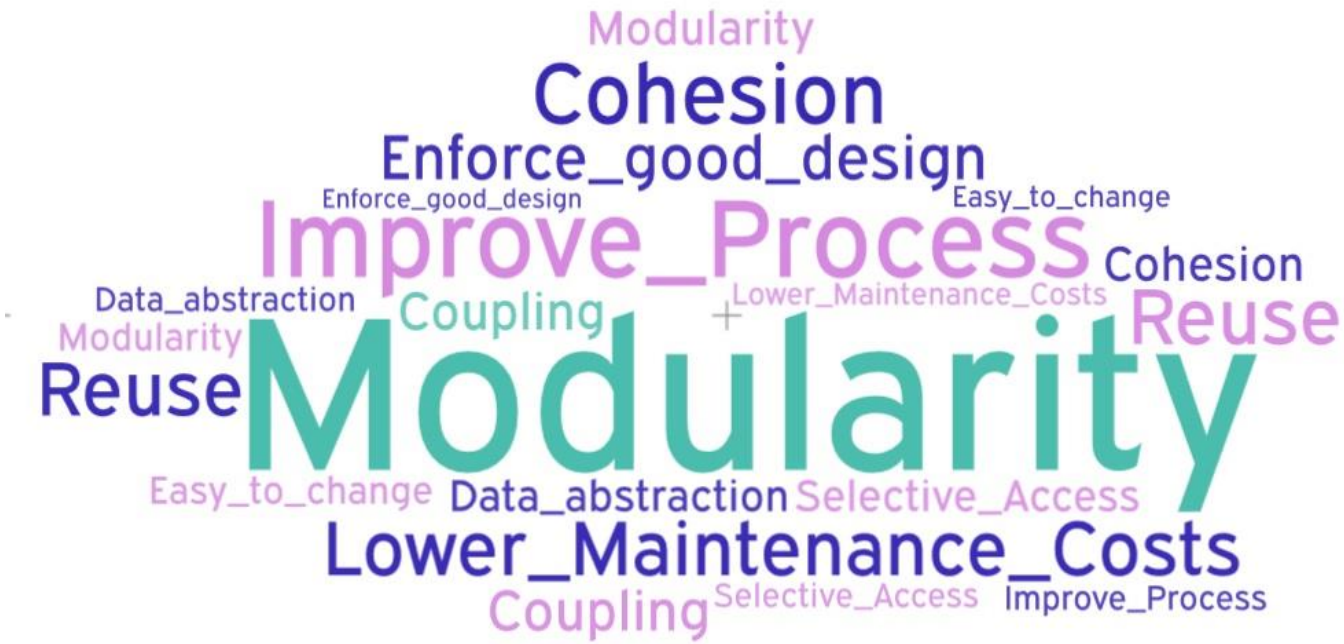
Week10: Intermediate C++

Recap: Object Orientation

- What does it mean to think in terms of object orientation?
  1. Give precedence to data over functions *(think: objects, attributes, methods)*
  2. Hide information under well-defined and stable interfaces *(think: encapsulation)*
  3. Enable incremental refinement and (re)use *(think: inheritance and polymorphism)*

Object Orientation: Why?

- Improve costs
- Improve development process and
- Enforce good design



Objects and Instances

- Object is a computational unit
  - Has a state and operations that operate on the state.
  - The state consists of a collection of *instance* variables or attributes.
  - Send a “message” to an object to invoke/execute an operation (*message-passing metaphor* in traditional OO thinking)
- An instance is a *specific version* of the object

Classes

- Template or blueprint for creating objects. Defines the shape of objects
  - Has *features* = attributes + operations
  - New objects created are *instances of the class*
  - E.g.

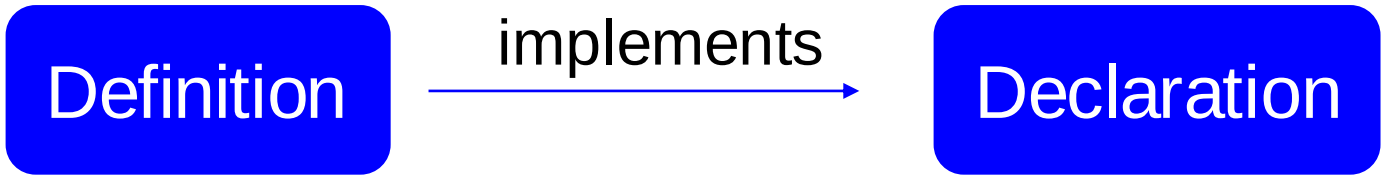
Class - lollypop mould

Objects - lollypops

Classes continued..

- Operations defined in a class are a prescription or service provided by the class to access the state of an object
- Why do we need classes?
  - To define user-defined types / invent new types and extend the language
  - Built-in or Primitive types of a language – int, char, float, string, bool etc. have implicitly defined operations:
    - E.g. cannot execute a *shift* operator on a negative integer
  - Composite types *(read: classes)* have operations that are implicit as well as those that are explicitly defined.

Classes declaration vs. definition



Implementation of functions in a .cpp file

listing of functions and attributes in a .h file

Classes: declaration

- *file* `Fruit.h`

```
#include<string>

class Fruit {
      string commonName;

public:
      Fruit(string name);
      string GetName();
};
```

Common terms for the state of an object:
“fields”, “attributes”, “property”, “data”
“characteristic”

Class Name

Attribute

Constructor

Common terms for operations:
“functions”, “behavior”, “message”,
“methods”, “responsibilities”

Method

Classes: access control

- Public / Private / Protected

```
class Fruit {
        string commonName; // private by default

public:
        Fruit(string name);
        string GetName();
};
```

- Private: methods-only (self) access
- Public: all access
- Protected: methods (self and *sub-class*) access

Friend functions

- Can access `private` and `protected` members

```
class Coconut {
        vector<pair<string, float> > constituents;
public:
        ...
        friend float ComputeEnergy(float wt, Coconut* c);
};

float ComputeEnergy(float weight, Coconut* c) {
//get a set of items, for each item, get its weight and
//energy_per_g. multiply both. Sum the product of all items...
//read from c->constituents to get the set of items.
}
```

The non-member function `ComputeEnergy` can access private attribute `constituent` of `Coconut` class

Classes: definition

- *file* `Fruit.cpp`

```
#include<Fruit.h>

//constructor definition: initialize all attributes
Fruit::Fruit(string name) {
	commonName = name;
}
//constructor definition can also be written as:
Fruit::Fruit(string name): commonName(name) { }

string Fruit::GetName() {
	return commonName;
}
```

Objects: creation and usage

- *file* `Fruit.cpp`

```
#include<Fruit.h>

Fruit::Fruit(string name): commonName(name) { }
string Fruit::GetName() { return commonName; }

int main() {
        Fruit obj1("Mango"); //calls constructor
        //following line prints "Mango"
        cout<<obj1.GetName()<<endl; //calls GetName method
}
```

- *How is `obj1` destroyed?* – by calling *destructor*

Objects: Destructor

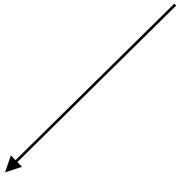
```
Fruit::~Fruit(){ } //default destructor implicitly defined

int main() {
        Fruit obj1("Mango"); //statically allocated object
        Fruit* obj2 = new Fruit("Apple"); //dynamic object
        delete obj2; //calls obj2->~Fruit();
        //calls obj1.~Fruit()
}
```

- Statically allocated objects: Automatic
- Dynamically allocated objects: Explicit

Inheritance

- Create a brand-new class based on existing class

```
file Mango.h
#include<Fruit.h>
class Mango : public Fruit {
        string variety;
public:
        Mango(string name, string var) : Fruit(name),
variety(var){}
};
```



- Fruit is a base type, Mango is a sub-type
- Sub-type inherits attributes and methods of its base type

Inheritance

```
file Fruit.h
#include<string>

class Fruit {
        string commonName;
public:
        Fruit(string name);
        string GetName();
};
```

```
file Mango.h
#include<Fruit.h>
class Mango : public Fruit {
        string variety;
public:
        Mango(string name, string var) :
Fruit(name), variety(var){}
};
```

```
file Fruit.cpp
...
int main() {
        Mango item1("Mango", "Alphonso"); //create sub-class object
        cout<<item1.GetName()<<endl; //only commonName is printed!
}                                    (variety is not included).
```

commonName variety

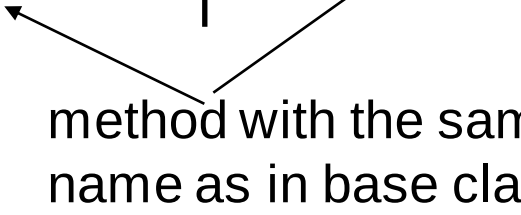
Refer slide 41.

Method overriding

- Customizing methods of derived / sub- class

```
file Fruit.h
#include<string>

class Fruit {
        string
commonName;
public:
        Fruit(string
name);
        string GetName();
};
```

```
file Mango.h
#include<Fruit.h>
class Mango : public Fruit {
        string variety;
public:
        Mango(string name, string var) :
Fruit(name), variety(var){}
    string GetName();
};
```



Method overriding

```
file Fruit.h
#include<string>

class Fruit {
protected:
        string commonName;
public:
        Fruit(string name);
        string GetName();
};
```

```
file Mango.h
#include<Fruit.h>
class Mango : public Fruit {
        string variety;
public:
        Mango(string name, string var) :
Fruit(name), variety(var){}
        string GetName() {    return
commonName + "_" + variety; }
};
```


accessing base
class attribute

Method overriding

file Fruit.h

```
#include<string>

class Fruit {
protected:
        string commonName;
public:
        Fruit(string name);
        string GetName();
};
```

file Mango.h

```
#include<Fruit.h>
class Mango : public Fruit {
        string variety;
public:
        Mango(string name, string var) :
Fruit(name), variety(var){}
        string GetName() {    return
commonName + "_" + variety; }
};
```

file Fruit.cpp

```
...
int main() {
        Mango item1("Mango", "Alphonso"); //create sub-class object
        cout<<item1.GetName()<<endl;   //prints "Mango_Alphonso"
}
```

Polymorphism

- Ability of one type to appear and be used as another type
- E.g. type `Mango` used as type `Fruit`

```
file Fruit.cpp
...
int main() {
//create a sub-class object and initialize it to a pointer of
//type base-class
        Fruit* item1 = new Mango("Mango", "Alphonso");
        cout<<item1->GetName()<<endl;  //prints "Mango" !
        ...
}
```

Trivia: Java treats all functions as virtual

Polymorphism

- Declare overridden functions as `virtual` in base class
- Invoke those functions using pointers

file Fruit.h

```
#include<string>

class Fruit {
protected:
        string commonName;
public:
        Fruit(string name);
        virtual string GetName();
};
```

file Mango.h

```
#include<Fruit.h>
class Mango : public Fruit {
        string variety;
public:
        Mango(string name, string
var) : Fruit(name), variety(var){}
string GetName() {    return
commonName + "_" + variety; }
};
```

```
Fruit* item1 = new Mango("Mango", "Alphonso");
cout<<item1->GetName()<<endl; //prints "Mango_Alphonso"
```

Polymorphism and Destructors

- declare base class destructors as `virtual` if using base class in a polymorphic way

```
file Fruit.h
#include<string>

class Fruit {
protected:
        string commonName;
public:
        Fruit(string name);
        virtual string GetName();
        virtual ~Fruit();
};
```

```
...
Fruit* item1 = new Mango("Mango",
"Alphonso");
...
delete item1; //calls Mango::~Mango()
first and then Fruit::~Fruit()
```