

CS601: Software Development for Scientific Computing

Autumn 2024

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



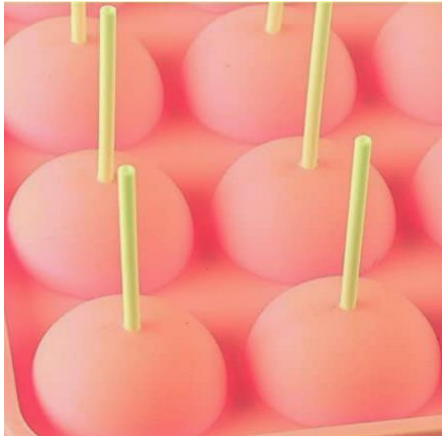
© Nikhil Hegde 2020

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

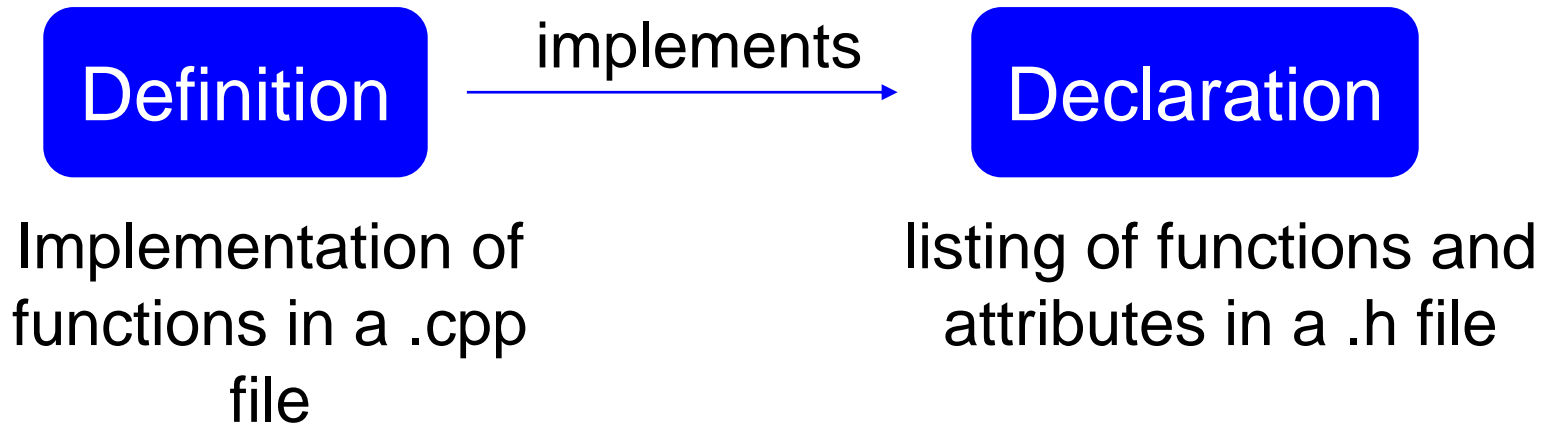


Objects - lollypops

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



Classes: declaration

- *file* Fruit.h
#include<string>

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

```
class Fruit {  
    string commonName; } Attribute
```

Class Name

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

Constructor

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

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 constituents 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){}
};
```

calling base-class
constructor



- 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).
```

```
}  
Nikhil Hegde
```

```
Refer slide 41.
```

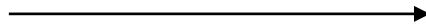
Inheritance – Access Control

What is available from Base and [how](#)

Base

Private
Protected
Public

Public inheritance



Derived

protected:
Base::Protected
public:
Base::Public

Base

Private
Protected
Public

Private inheritance



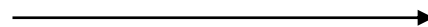
Derived

Private:
Base::Protected
Base::Public

Base

Private
Protected
Public

Protected inheritance



Derived

Protected:
Base::Protected
Base::Public

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 with the same
name as in base class

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 Fruit.cpp
```

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

```
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; }  
};
```

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()
```

Abstract base classes

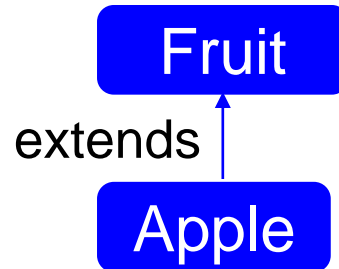
- A class can have a virtual method without a definition – *pure virtual functions*

- E.g

```
class Fruit {
protected:
    string commonName;
    float weight;
    float energyPerUnitWeight; //in kCals / 100g
public:
    Fruit(string name, float weight);
    virtual string GetName();
    virtual ~Fruit();
    virtual void Energy() = 0;
};
```

Energy is 'pure' – no implementation

Defining pure virtual function

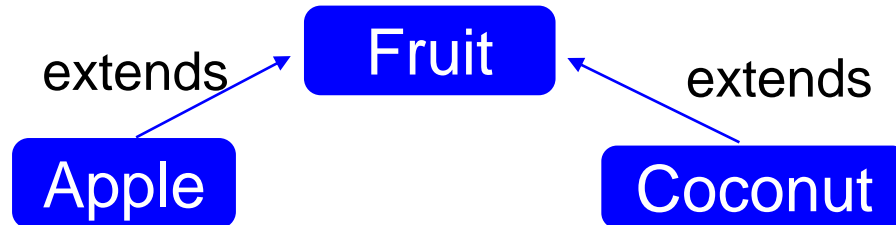


```
class Apple : public Fruit {
    vector<pair<string, float> > constituents;
public:
    Apple(string name, float weight);
    virtual ~Apple();
    . . .
    void Energy() {
        energyPerUnitWeight = ComputeEnergy(weight, constituents);
    }
};
```

Pure virtual method
defined in derived class.

Base class attribute

Defining pure virtual function



```
class Coconut : public Fruit {  
    vector<pair<string, float> > constituents;  
public:  
    Coconut(string name, float weight);  
    virtual ~Coconut();  
    . . .  
    void Energy() {  
        float effWeight = GetEdibleContentWeight();  
        energyPerUnitWeight = ComputeEnergy(effWeight, constituents);  
    }  
};
```

Computation is different from that of Apple's method

Abstract base classes..

- Cannot create objects from abstract base classes. But may need constructors. Why?

```
Fruit item1; //not allowed. Fruit::Energy() is pure virtual
```

- Can create pointers to abstract base classes and use them in polymorphic way

```
Fruit* item1 = new Apple("Apple", 0.24);  
cout<<item1->Energy()<<"Kcals per 100 g"<<endl;
```

- Often used to create *interfaces*

Operator overloading

- How can we assign one object to another?

```
Apple a1("Apple", 1.2); //constructor Apple::Apple(string, float)
                          //is invoked
```

```
Apple a2; //constructor Apple::Apple() is invoked.
```

```
a2 = a1 //object a1 is assigned to a2. assignment operator invoked
```

```
Apple& Apple::operator=(const Apple& rhs) {
    commonName = rhs.commonName;
    weight = rhs.weight;
    energyPerUnitWeight = rhs.energyPerUnitWeight;
    constituents = rhs.constituents;
    return *this;
}
```

Called Copy Assignment Operator

Another Example

Header file (myvec.h)

```
#ifndef MYVEC_H  
#define MYVEC_H  
  
#endif
```

Header file (myvec.h)

- Declare the class

Class *declaration* opening scope

Keyword

Class name

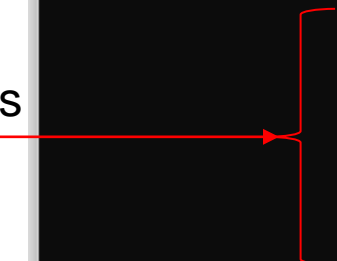
Class *declaration* closing scope

```
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
};
#endif
```

Header file (myvec.h)

```
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
    //private attributes
    double* data;
    int vecLen;
};
#endif
```

Declaring attributes



Header file (myvec.h)

```
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
    //private attributes
    double* data;
    int vecLen;
public:
    MyVec(int len); //constructor
    ~MyVec(); //destructor
};
#endif
```

Specifying access control



Declaring operations



Defining the class (myvec.h and myvec.cpp)

```
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
    //private attributes
    double* data;
    int vecLen;
public:
    MyVec(int len); //constructor decl.
    ~MyVec(); //destructor decl.
};
#endif
```

```
#include "myvec.h"
//defining the constructor
MyVec::MyVec(int len) {
    vecLen=len;
    data=new double[vecLen];
}
//defining the destructor
MyVec::~~MyVec() {
    delete [] data;
}
```

Scope resolution operator
Constructor: no return type.
Destructor: no parameters, no return type.

Defining the class (myvec.h and myvec.cpp)

```
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
    //private attributes
    double* data;
    int vecLen;
public:
    MyVec(int len); //constructor decl.
    ~MyVec(); //destructor decl.
    int GetVecLen(); //member function
};
#endif
~
~
```

```
#include "myvec.h"
//defining the constructor
MyVec::MyVec(int len) {
    vecLen=len;
    data=new double[vecLen];
}
//defining the destructor
MyVec::~~MyVec() {
    delete [] data;
}
//defining GetVecLen member function
int MyVec::GetVecLen() {
    return vecLen;
}
```

Operator overloading []

```
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
    //private attributes
    double* data;
    int vecLen;
public:
    MyVec(int len); //constructor decl.
    ~MyVec(); //destructor decl.
    int GetVecLen(); //member function
    double& operator[](int index);

};

delete [] data;
}
//defining GetVecLen member function
int MyVec::GetVecLen() {
    return vecLen;
}
double& MyVec::operator[](int index) {
    return data[index];
}

#endif
```

Operator overloading - usage

```
#include<iostream>
#include"myvec.h"
using namespace std;
int main() {
    MyVec v(10); //calls the constructor MyVec::MyVec(int) and passes the argument 10
    int size=v.GetVecLen(); //calls the member function
    cout<<"size of MyVec is: "<<size<<" elements"<<endl;
    cout<<"Setting first element to 100"<<endl;
    v[0]=100;
    cout<<"Fetching first element value: "<< v[0] << endl;
}
```

Copying Objects

```
Apple a1("Apple_red", 0.2);  
Apple a2 = a1; //calls copy constructor
```

```
Apple::Apple(const Apple& rhs) {  
    commonName = rhs.commonName;  
    weight = rhs.weight;  
    energyPerUnitWeight = rhs.energyPerUnitWeight;  
}
```

Copy constructor - usage

```
#include<iostream>
#include"myvec.h"
using namespace std;
int main() {
    MyVec v(10); //calls the constructor MyVec::MyVec(int) and passes the argument 1
    int size=v.GetVecLen(); //calls the member function
    cout<<"size of MyVec is: "<<size<<" elements"<<endl;
    cout<<"Setting first element to 100"<<endl;
    v[0]=100;
    cout<<"Fetching first element value: "<< v[0] << endl;
    MyVec v2=v; //calls the copy constructor
    cout<<"v2's first element: "<<v2[0]<<endl;
}
```

- Not necessary to define the copy constructor. Compiler defines one for us.

```

#include<iostream>
#include"myvec.h"
using namespace std;
int main() {
    MyVec v(10); //calls the constructor MyVec::MyVec(int) and passes the argument 10
    int size=v.GetVecLen(); //calls the member function
    cout<<"size of MyVec is: "<<size<<" elements"<<endl;
    cout<<"Setting first element to 100"<<endl;
    v[0]=100;
    cout<<"Fetching first element value: "<< v[0] << endl;
    MyVec v2=v; //calls the copy constructor
    cout<<"v2's first element: "<<v2[0]<<endl;
}

```

```

size of MyVec is: 10 elements
Setting first element to 100
Fetching first element value: 100
v2's first element: 100
free(): double free detected in tcache 2
Aborted

```

```

#include<iostream>
#include"myvec.h"
using namespace std;
int main() {
    MyVec v(10); //calls the constructor MyVec::MyVec(int) and passes the argument 10
    int size=v.GetVecLen(); //calls the member function
    cout<<"size of MyVec is: "<<size<<" elements"<<endl;
    cout<<"Setting first element to 100"<<endl;
    v[0]=100;
    cout<<"Fetching first element value: "<< v[0] << endl;
    MyVec v2=v; //calls the copy constructor
    cout<<"v2's first element: "<<v2[0]<<endl;
}

```

```
size of MyVec is: 10 elements
```

```
Setting first element to 100
```

If you don't define a copy constructor, in some cases, e.g., for class MyVec, the program aborts. Why in this case?

```
v2's first element: 100
```

```
free(): double free detected in tcache 2
```

```
Aborted
```


const and references

```
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
    //private attributes
    double* data;
    int vecLen;
public:
    MyVec(int len); //constructor decl.
    MyVec(const MyVec& rhs); //copy constructor decl.
    int GetVecLen() const; //member function decl.
    double& operator[](int index) const;
    ~MyVec(); //destructor decl.
};

}

MyVec::MyVec(const MyVec& rhs) {
    vecLen=rhs.GetVecLen();
    data=new double[vecLen];
    for(int i=0;i<vecLen;i++) {
        data[i] = rhs[i];
    }
}

//defining GetVecLen member function
int MyVec::GetVecLen() const {
    return vecLen;
}

double& MyVec::operator[](int index) const {
    return data[index];
}
```

```

#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
    //private attributes
    double* data;
    int vecLen;
public:
    MyVec(int len); //constructor decl.
    MyVec(const MyVec& rhs); //copy constructor decl.
    int GetVecLen() const; //member function decl.
    double& operator[](int index) const; //operator decl.
    ~MyVec(); //destructor decl.
};

MyVec::MyVec(const MyVec& rhs) {
    vecLen=rhs.GetVecLen();
    data=new double[vecLen];
    for(int i=0;i<vecLen;i++) {
        data[i] = rhs[i];
    }
}

//defining GetVecLen member function
int MyVec::GetVecLen() const {
    return vecLen;
}

double& MyVec::operator[](int index) const {
    return data[index];
}

```

Define the copy constructor. Now you need to make changes to other methods (const) as well.

```

Setting first element to 100
Fetching first element value: 100
v2's first element: 100

```

Returning References- Example1

- How can we assign one object to another?

```
Apple a1("Apple", 1.2); //constructor Apple::Apple(string, float)
                        //is invoked
```

```
Apple a2; //constructor Apple::Apple() is invoked.
```

```
a2 = a1 //object a1 is assigned to a2;assignment operator is invoked
```

```
Apple& Apple::operator=(const Apple& rhs)
```

Called Copy Assignment Operator

```
Apple& Apple::operator=(const Apple& rhs) {
    commonName = rhs.commonName;
    weight = rhs.weight;
    energyPerUnitWeight = rhs.energyPerUnitWeight;
    constituents = rhs.constituents;
    return *this;
}
```

What is Move Assignment Operator?

this

- Implicit variable defined by the compiler for every class
 - E.g. `MyVec *this;`
- All member functions have `this` as an implicit first argument
 - E.g.
`int MyVec::GetVecLen() const;`
would actually be:
`int MyVec::GetVecLen(MyVec* this) const;`

Overloading +=

- `MyVec v1;`
`v1+=3;`
- `MyVec& MyVec::operator+=(double)`

Overloading +=

- `MyVec v1;`
`v1+=3;`
 - `MyVec& MyVec::operator+=(double)`
- `MyVec v2;`
`v2+=v1;`
 - `MyVec& MyVec::operator+=(const MyVec& rhs)`
 - What if you make the return value above `const`?
 - Disallow: `(v2+=v1)+=3;`

Overloading +

- `v1=v1+3;` *Single-argument constructors: allow implicit conversion from a particular type to initialize an object.*
 - `const MyVec MyVec::operator+(double val)`
- `v3=v1+v2;`
 1. `const MyVec MyVec::operator+(const MyVec& vec2) const;`

OR

2. `friend const MyVec operator+(const MyVec& lhs, const MyVec& rhs);`

`v1=3+v1` is compiler error! Why?

Operator Overloading - Guidelines

- If a binary operator accepts operands of different types and is commutative, both orders should be overloaded
- Consistency:
 - If a class has `==`, it should also have `!=`
 - `+=` and `+` should result in identical values
 - define your copy assignment operator if you have defined a copy constructor