Object-Oriented Programming and Classes

Self Declaration

- The content is exclusively meant for academic purpose and for enhancing teaching and learning. Any other use of economic /commercial purpose is strictly prohibited. The users of the content shall not distribute, disseminate or share it with anyone else and its use is restricted to advancement of individual knowledge .The information provided in this e-content is authentic and best as per my knowledge.

Mr.

Vijay kant Sharma

Department of computer application

Jagatpur P.G. College Varanasi

Motivation

Basic, built-in, pre-defined types: char, int, double, …

Variables + operations on them

```
int a, b,c;
c=a+b;
c=a mod b;
…
```

More complicated, user-defined types: classes

Variables → objects
Types → classes

procedural programming:
a sequence of ‘procedures’

```
int main()
{
  int x,y,z;
  int a,b,c;


        a=f1(x);
        b=f2(y);
        c=f3(z);
        …

}

int f1()
{
}

int f2()
{
}

int f3()
{
}
```

```
int main()
{
  A a;
  B b;
  C c;

  a.f1();
  b.f2();
  c.f3();
   …
}

Class A
{
Int x;
Int f1();
}

Class B
{
Int y;
Int f2()
}

Class C
{
Int z;
Int f3();
}
```

Object oriented programming:
a sequence of ‘objects’!

Motivation

Variables → objects
Types → classes

Procedural programming:

- Low-level, closer to hardware
- More intuitive, less abstract
- More ‘action’ oriented
- Focus on ‘action’, ‘procedure’, ‘method’
- Procedure-oriented

Object-oriented programming:

- High-level
- More abstract
- Focus on ‘what to do’ not on ‘how to do’

In the implementation of OOP,
we still need sound ‘procedure programming’ skills!

Motivation

We want to build user-defined (and “smart”) objects that can answer many questions (and perform various actions).

- “What is your temperature?”
- “What is your temperature in Fahrenheit?”
- “What is your humidity?”
- “Print your temperature in Celsius.”

Temperature example

- Write a program that, given a temperature in Fahrenheit or Celsius, will display the equivalent temperature in each of the scales.

```
double degree = 0.0;    // needs 2 items!
char scale = 'F';
```

- To apply a function `f()` to a temperature, we must specify both degree and scale:

```
f(degree, scale);
```

- Also to display a temperature:

```
cout << degree << scale;
```

Put related variables together …

(remember that an Array is a collection of variables of same type)

The simpliest Class (or a C-structure) can be thought of being a collection of variables of different types

A first simple ‘class’ or ‘object-oriented’ solution

```
class Temperature {
    public:
      double degree;
      char scale;
};
```

Two member variables: degree **and** scale

a new (user-defined) type, a composite type: Temperature!

Remark:

In old C, this can be done using ‘structure’:

similar to ‘record’ in Pascal

```
structure Temperature {
      double degree;
      char scale;
};
```

The dot operator for (public) members

The modifier 'public' means that the member variables can be accessed from the objects, e.g.

```
Temperature temp1, temp2;

temp1.degree=54.0;
temp1.scale='F';
temp2.degree=104.5;
temp2.scale='C';
```

A C++ struct is a class in which all members are by default public.

Manipulation of the new type:

Some basic operations:

```
void print(Temperature temp) {
        cout << "The temperature is degree "
                << temp.degree <<  "with the scale " <<
                 temp.scale << endl;
}
double celsius(Temperature temp) {
     double cel;
     if (temp.scale=='F') cel=(temp.degree-32.0)/1.8;
     else cel=temp.degree;
     return cel;
}
double fahrenheit(Temperature temp) {
   double fa;
   if(temp.scale=='C') fa= temp.degree *1.8+32.0;
   else fa=temp.degree;
   return fa;
}
```

An application example:

```
    Temperature annualtemp[12];


double annualAverageCelsius(Temperature arraytemp[]) {
   double av=0.0;
   for (int i=0;i<12;i++) av=av+celsius(arraytemp[i]);
   return av;
};
```

Put the variables and functions together …

Actual problem:

1. **Member 'variables' are still separated from 'functions' manipulating these variables.**
2. **However, 'functions' are intrinsically related to the 'type'.**

The simplest class (or a C-structure) defined this way is a collection of (member) variables (similar to RECORD in Pascal)

A more advanced class is a collection of (member) variables and (member) functions

"The art of programming is the art of organising complextity."

An improved Temperature class with member functions associated

Assembly the data and operations together into a class!

```
class Temperature{
      public:
            void print();       // member functions
            double celsius();
            double fahrenheit();

            double degree;      // member variables
            char scale;
};
```

Operators for members

The dot operator not only for public member variables of an object, but also for public member functions (during usage), e.g.

```
Temperature temp1;
temp1.celsius();
temp1.print();
```

function → method

Function(procedure) call → message

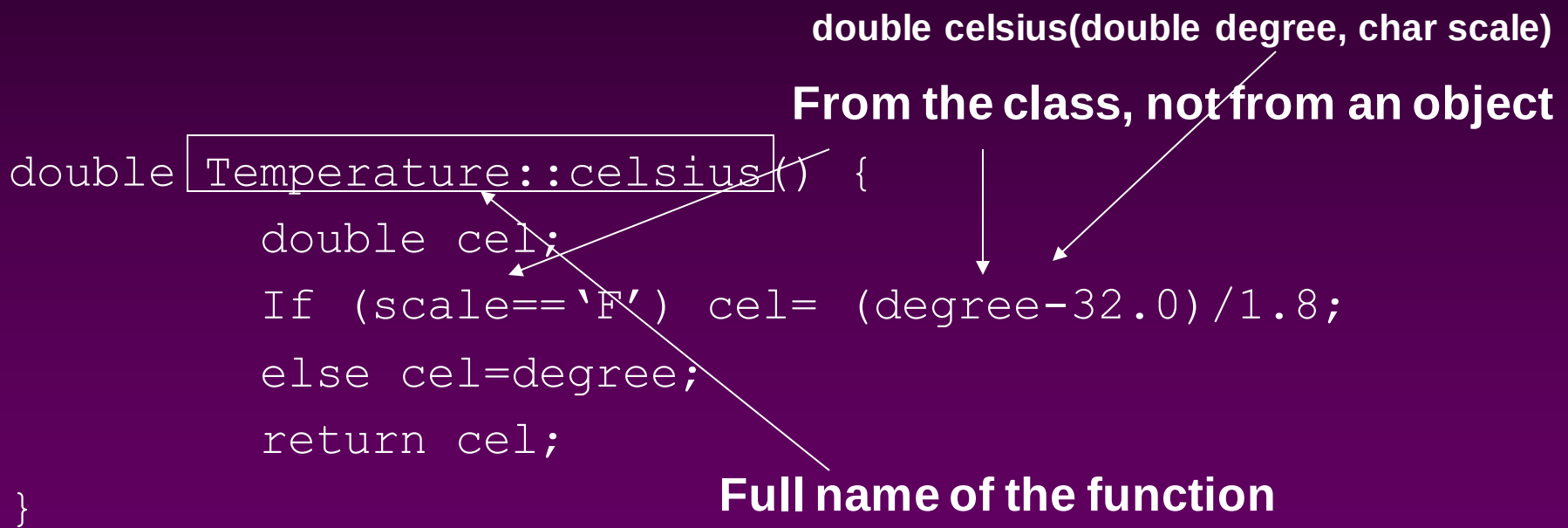
Comments:

1. Temp1 receives `print()` message and displays values stored in degree and scale, receives celsius() message to give the temperature in celsius …
2. It is not the function which is calling the object like

   print(temp1) traditionally, temp1.print() → object oriented!
3. The temperature are ‘smart objects’ ☺ unlike ‘stupid’ basic type objects

Operators for defining member functions

:: for member functions of a class (during definition)



:: is used with a class name while dot operator is with an object!

Using ‘private’ modifier!

‘private’ members can only be used by member functions, nothing else!

- **Using private member variables for data protection and information hiding**
- **Using member functions to access the private data instead**

- **Try to make functions ‘public’**
- **Try to make data ‘private’**

Global

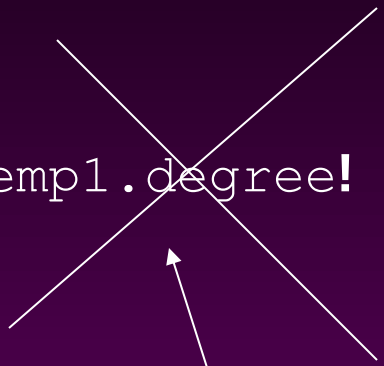
Local

New version of Temperature class

```
class Temperature{
            public:   // member functions
                void print();
                 double celsius();
                 double fahrenheit();
             private:  // member variables
                double degree;
                char scale;
        };
```

When the datum 'degree' is private,

it can not be accessed directly by using `temp1.degree`**!**



```
double Temperature::celsius() {
        double cel;
        If (scale=='F') cel= (degree-32.0)/1.8;
        else cel=degree;
        return cel;
}
```

Possible only when 'degree' is public or in member functions

OK

Private member variables can only be accessed by 'member functions' of the same class.

Using member functions to (indirectly) access private data

```
class Temperature{
            public:   // member functions
                 double getDegree();
                 char getScale();
                 void set(double newDegree, char newScale);

                 void print();
                  double celsius();
                  double fahrenheit();
           private:  // member variables
                    double degree;
                    char scale;
      };
```

Some member functions on private data:

```
double Temprature::getDegree() {
    return degree;
}

double Temprature::getScale() {
    return scale;
}

double Temprature::set(double d, char s) {
  degree = d;
  scale = s;
}
```

(Temporary) Summary:

- A collection of member variables and member functions is a Class
- Struct is a class with only member variables, and all of them public
- ‘public’ member can be used outside by dot operator
- ‘private’ member can only be used by member functions
- Dot operator for objects and Scope resolution operator :: for class

```
class A {
   public:
        void f();

        int x;
   private:
        int y;
}

void A::f() {
   x=10;
   y=100;
}
```

```
int main() {
   A a;

   a.f();

   cout << a.x << endl;
   cout << a.y << endl; // no!!!

   a.x = 1000;
   a.y = 10000; // no!!!
}
```

Some basic member functions:

Classification of member functions:

- **Constructors for initialisation**
- **Access for accessing member variables**
- **Update for modifying data**
- **I/O and utility functions …**

The other (application) functions should be built on these member functions!

A more complete definition

A complete class should have a complete set of basic member functions manipulating the class objects

```
class Temperature{
      public:
            Temperature();
            Temperature(double idegree, char iscale);
            double getDegree() const;
            char getScale() const;
            void set(double newDegree, char newScale);
            void read();
            void print() const;
            double fahrenheit();
            double celsius();
      private:
            double degree;
            char scale;
};
```

Protection of data: ‘const’ modifier

Default-Value Constructor

A constructor is a special member function whose name is always the same as the name of the class.

```
Temperature::Temperature(){
   degree = 0.0;
   scale = 'C';
}
```

A constructor function initializes the data members when a Temperature object is declared.

```
Temperature temp3;
```

Remarks on ‘constructor’:

- Constructor functions have
  no return type (not even void!).
- Because a constructor function initializes the data members, there is no `const` following its heading.
- ‘constructor’ is over-loaded
- The constructor function is automatically called whenever a Temperature class object is declared.

Application of Temperature class

```
#include <iostream>
using namespace std;

// definition of Temperature class goes here

void main(){
   char resp;
   Temperature temp;
   do{
       cout << "Enter temperature (e.g., 98.6 F): ";
       temp.read();
       cout << temp.fahrenheit() << “Fahrenheit” << endl;
       cout << temp.celsius() << “Celsius” << endl;
       cout << endl << endl;
       cout << "Another temperature to convert? ";
       cin >> resp;
   }while(resp == 'y' || resp == 'Y');
}
```

‘Smart’ Temperature Object

- A smart object should carry within itself the ability to perform its operations
- Operations of `Temperature` object :
  - initialize degree and scale with default values
  - read a temperature from the user and store it
  - compute the corresponding Fahrenheit temperature
  - compute the corresponding Celsius temperature
  - display the degrees and scale to the user

Overloading

function overloading

```
#include <stdio.h>

  int max(int a, int b) {
    if (a > b) return a;
    return b;
  }

  char* max(char* a, char* b) {
    if (strcmp(a, b) > 0) return a;
    return b;
  }

  int main() {
    printf("max(19, 69) = %d\n", max(19, 69));
// cout << "max(19, 69) = " << max(19, 69) << endl;
    printf("max(abc, def) = %s\n", max("abc", "def"));
// cout << "max("abc", "def") = " << max("abc", "def") << endl;
    return 0;
  }
```

Operator Overloading

To enable conventional notations:

```
class Complex {
    ...
  public:
    ...
    Complex operator+(const Complex& op) {
      double real = _real + op._real,
             imag = _imag + op._imag;
      return(Complex(real, imag));
    }
    ...
  };
```

- An expression of the form

```
c = a + b;
```

- is translated into a method call

```
c = a.operator+(a, b);
```

- The overloaded operator may not be a member of a class: It can rather defined outside the class as a normal overloaded function. For example, we could define operator + in this way:

```
class Complex {
    ...
  public:
    ...
    double real() { return _real; }
    double imag() { return _imag; }

    // No need to define operator here!
  };
  Complex operator+(Complex& op1, Complex& op2) {
    double real = op1.real() + op2.real(),
           imag = op1.imag() + op2.imag();
    return(Complex(real, imag));
  }
```

Generic programming and templates

It's important to know it, though we don't ask you to 'do' it.

Templates

A function or a class or a type is parameterized by another type T.

Template or not template, that's the question of 171!

Function templates

```
int    max(const int&    a, const int&    b) {…}
double max(const double& a, const double& b) {…}
string max(const string& a, const string& b) {…}
```

A parameter T is a type

```
T      max(const T&      a, const T&      b) {…}
```

```
template<typename T>
T max(const T& a, const T& b)
{
 if (a>b) return a; else return b;
}
```

Also ‘template <class T>’

Function templates:
not functions, patterns or templates

```
Main()
{
int i=1,j=2;
cout << max(i,j);

string a(“hello”), b(“world);
cout << max(a,b);
}
```

Compiler creates two max() functions: template instantiation

Class templates

```
template <typename T>
class A {
public:
   A(T data);
  ~A() { delete x;}
  T get() {return *x;}
  T set(T data);
private:
  T* x;
};

template<typename T>
A<T>::A(T data)
{
X=new; *x=data;
}
tempalte<typename T>
T A<T>::set(T data)
{ *x=data; }
```

```
#include <iostream>
#include <string>
using namespace std;

main()
{
A<int> i(5);
A<string> s("comp171");

Cout << i.get() << endl;
Cout << s.get() << endl;

i.set(10);
s.set("CS");
cout << i.get() << endl;
cout << s.get() << endl;
```

Class templates: STL—Standard Template Library

- C type arrays → vectors and strings
- Problems with arrays:
  - Not copied with =
  - No size
  - No index checking
- We learnd the C++ string class
- Vector size is variable …

```
#include <iostream>
#include <vector>
using namespace std;

main()
{

vector<int> A(100);

For (int i=0; i<A.size();i++) {
        A[i] = …;
}

}
```

An example of generic linked List

```
template<typename T>
class Node {
public:
  T data;
  Node<T>*  next;
}

template<typename T>
class List {
Public:
…
Private:
   Node<T>* head;
}
```



```
struct Node{
     public:
            int data;
            Node* next;
};
typedef Node* Nodeptr;

class listClass {
     public:
            listClass();                    // constructor
            listClass(const listClass& list1); // copy construc
            ~listClass();                   // destructor

            bool empty() const;             // boolean function
            int headElement() const;        // access functions

            void addHead(int newdata);      // add to the head
            void delHead();                 // delete the head

            int length() const;            // utility function
            void print() const;            // output

     private:
            Nodeptr head;
};
```

Separate compilation:

***.h interfaces**

Few in procedural programming

Lots in OOP

***.cc implementations**

Fewer in OOP

Other advanced class concepts:

Sub-class, Derived classes, and class hierarchies (comp151)

→ Polygon → line segment → point

Friend

- A friend of a class can access to its private data members

```
class Vector {
    friend Vector operator*(const Matrix&, const Vector&);
};

Class Matrix {
    friend Vector operator*(const Matrix&, const Vector&);
};
```

Summary

- A class can be used not only to combine data but also to combine data and functions into a single (compound) object.
- A member variable or function may be either public or private
  - It can be used outside of the class when it's public
  - It can only be used by other member functions of the same class when it's private
- An object of a class can be copied by "=", memberwise copy (for static classes)
- 'const' is part of the function definition
- A constructor is a member function called automatically when an object is declared
  - Each class has to have at least one constructor
  - Constructors can be overloaded as long as the argument lists are different

Abstract Data Type

What is an abstract data type?

A data type consists of a collection of values together with a set of basic operations on these values

A data type is an abstract data type if the programmers who use the type do not have access to the details of how the values and operations are implemented.

All pre-defined types such as int, double, … are abstract data types

An abstract data type is a 'concrete' type, only implementation is 'abstract'

Abstract Data Type

- An Abstract Data Type is a class with some special restrictions.
- These restrictions can make programming easier.
- One of these restrictions is called *information hiding*, used as black boxes, hide the implementation details
- In information hiding, the user should not be allowed to access the data members directly (they should be private).
- An Abstract Data Type is used in Object-Oriented Programming (COMP151).

How to do it in C++ with classes?

Separate the public interface from implementation,

If you change the implementation, you don't need to change the other parts of the programmes.

- **Make all the member variables private**
  → private data (implementation details)
- **Make member functions public**
  → public interface

Rational Review

- Rational number
  - Ratio of two integers: `a/b`
    - Numerator over the denominator
- Standard operations
  - Addition

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

  - Subtraction

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

  - Multiplication

$$\frac{a}{b} * \frac{c}{d} = \frac{ac}{bd}$$

  - Division

$$\frac{a}{b} / \frac{c}{d} = \frac{ad}{bc}$$

Rational Representation

- Represent a numerator and denominator with two `int` data members
  - `Numerator` and `Denominator`
  - Data members private (information hiding)
- Public arithmetic member functions
  - Rational addition, subtraction, multiplication, division
- Public relational member functions
  - Equality and less than comparisons

Rational Overview

```
class Rational {
  public:
      // for Rational member functions
      // for everybody (like "global" variables)
  private:
      // for Rational data members
      // like "local" variables
} ;
```

Rational Class

```
class Rational{
   public:
              // default-value constructor
        Rational();
              // explicit-value constructor
        Rational(int numer, int denom = 1);
              // arithmetic functions
        Rational Add(const Rational r) const;
        Rational Subtract(const Rational r) const;
        Rational Multiply(const Rational r) const;
        Rational Divide(const Rational r) const;
              // relational functions
        bool Equal(const Rational r) const;
        bool LessThan(const Rational r) const;
              // i/o functions
        void Display() const;
        void Get();
   private:   // data members
        int Numerator;
        int Denominator;
};
```

```
int i;
i=0;
int j;
j=10;
int k;
k=j;
```

```
int i(0);
int j(10);
int k(j);
```

main()

```
void main(){
  Rational r;
  Rational s;

  cout << "Enter two rationals(a/b): ";
  r.Get();
  s.Get();
  Rational t(r);

  Rational sum = r.Add(s);
  r.Display();
  cout << " + ";
  s.Display();
  cout << " = ";
  sum.Display(); cout << endl;
  Rational product = r.Multiply(s);
  r.Display();
  cout << " * ";
  s.Display();
  cout << " = ";
  product.Display();    cout << endl;
}
```

const

You can use `const` on user-defined types as usual:

```
const Rational OneHalf(1,2);
OneHalf.Display(); // no problem
OneHalf.Get();  // illegal: OneHalf is a const
```

Implementation of Rational class

Default-Value Constructor

```
// default-value constructor
Rational::Rational(){
      Numerator = 0;
      Denominator = 1;
}
```

- Example

```
      Rational r;        // r = 0/1
```

Explicit-Value Constructor

```
// explicit-value constructor
Rational::Rational(int numer, int denom){
      Numerator = numer;
      Denominator = denom;
}

Example
      Rational t1(2,3); // t1 = 2/3
      Rational t2(2);    // t2 = 2/1 = 2
```

Note: the prototype is Rational(int numer, int denom = 1);

Copy Constructor (automatic)

```
// copy constructor, automatically provided
Rational::Rational(const Rational& r) {
   Numerator = r.Numerator;
   Denominator = r.Denominator;
}
```

r.Numerator is possible because it's in a member function.

So the private parts can only be used by member functions.

Example

```
Rational t1(2,3); // t1 = 2/3
Rational t2(t1);  // t2 = 2/3
```

Note: very important concept, and it is AUTOMATIC for static classes!

Arithmetic Functions

```
Rational Rational::Add(const Rational r) const{
        int a = Numerator;
        int b = Denominator;
        int c = r.Numerator;
        int d = r.Denominator;
        Rational result(a*d + b*c, b*d);
        return result;
}
```

Example

```
        Rational t(1,2), u(3, 4);
        Rational v = t.Add(u);
```

```
Rational Rational::Multiply(const Rational r) const{
      int a = Numerator;
      int b = Denominator;
      int c = r.Numerator;
      int d = r.Denominator;
      Rational result(a*c, b*d);
      return result;
}
```

Example

```
      Rational t(1,2), u(3, 4);
      Rational v = t.Multiply(u);
```

```
Rational Rational::Subtract(const Rational r) const {
      int a = Numerator;
      int b = Denominator;
      int c = r.Numerator;
      int d = r.Denominator;
      Rational result(a*d - b*c, b*d);
      return result;
}
```

Example

```
      Rational t(1,2), u(3, 4);
      Rational v = t.Subtract(u);
```

```
Rational Rational::Divide(const Rational r) const{
      int a = Numerator;
      int b = Denominator;
      int c = r.Numerator;
      int d = r.Denominator;
      Rational result(a*d, b*c);
      return result;
}
```

Example

```
      Rational t(1,2), u(3, 4);
      Rational v = t.Divide(u);
```

Relational Functions

```
bool Rational::Equal(const Rational r) const{
      double a, b;
      a = double(Numerator)/Denominator;
      b = double(r.Numerator)/r.Denominator;
      if(a == b)
            return true;
      else
            return false;
}
```

Example

```
       if(s.Equal(t))
            cout << "They are the same!";
```

```
bool Rational::LessThan(const Rational r) const{
      double a, b;
      a = double(Numerator)/Denominator;
      b = double(r.Numerator)/r.Denominator;
      if(a < b)
            return true;
      else
            return false;
}
```

Example

```
   if(s.LessThan(t))
       cout << "The first is less than the second!";
```

I/O Functions

```
void Rational::Display() const{
    cout << Numerator << '/' << Denominator;
}
```

Example

```
      t.Display();
```

I/O Functions

```
void Rational::Get(){
      char slash;
      cin >> Numerator >> slash >> Denominator;
      if(Denominator == 0){
         cout << "Illegal denominator of zero, "
              << "using 1 instead" << endl;
         Denominator = 1;
      }
}
```

Example

```
      t.Get();
```

Rational Representation

- Member functions
  - Constructors
    - Default-value constructor

```
Rational r;
```

    - Explicit-value constructor

```
Rational r(3, 4);
```

    - Copy constructor (provided automatically: simply copies data members)

```
Rational r(t);  Rational r = t;
```

initialisation

  - Assignment (provided automatically: simply copies data members)

```
r = t;
```

assignment

  - Inputting and displaying object

ADT and class

- They are different concepts
- We use ‘class’ to implement the concept of ‘ADT’
- Class can do much more than ADT
- …