CS2311 Computer Programming

LT8: Classes & Objects

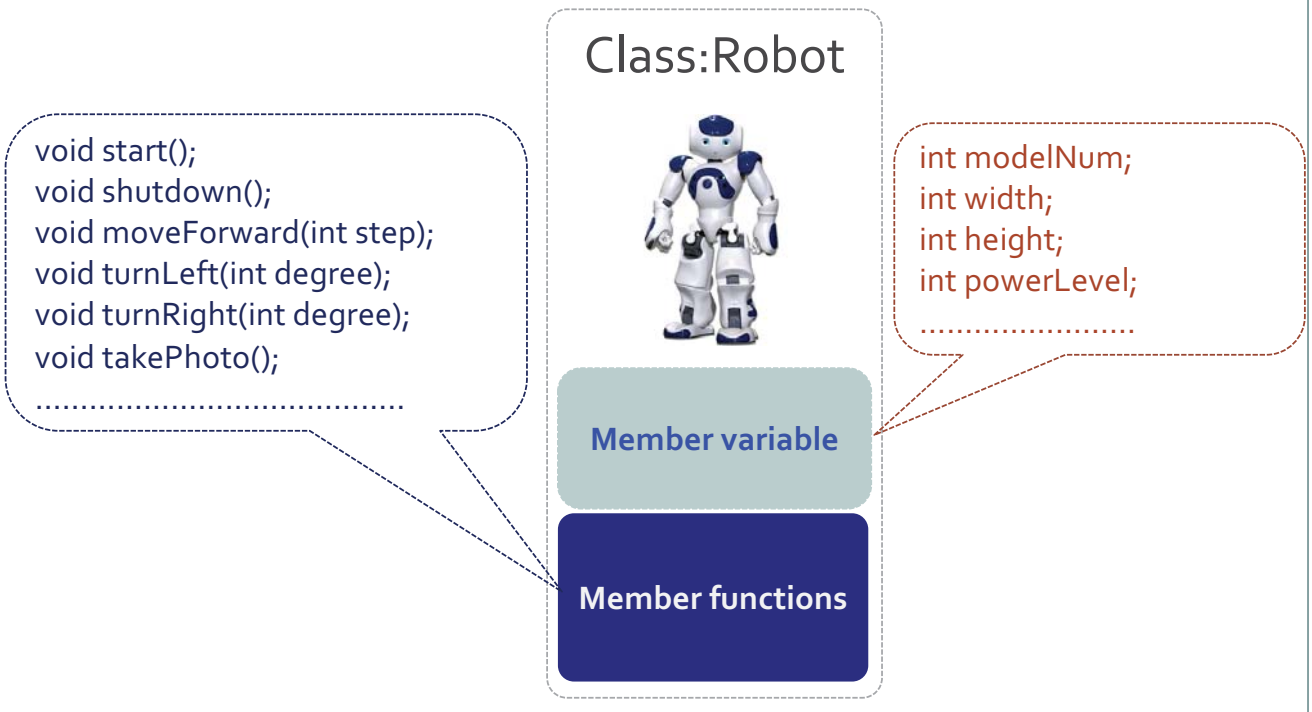
Outline

- Defining classes
- Defining member functions & scope resolution operator
- Public & private members
- Accessors
- Constructors
- *Friend functions*
- *Const modifier*
- *Operator overloading*
- *C-like Structure*

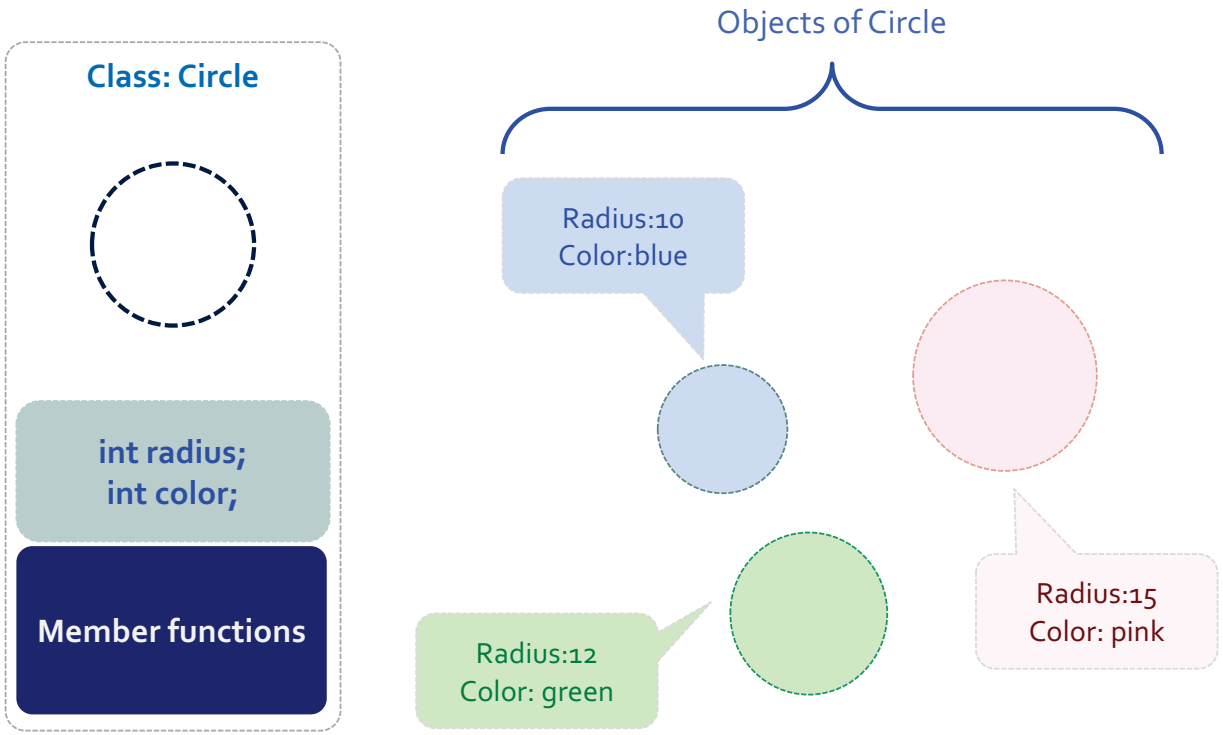
Class and Object

- Class and object are important features of Object-oriented Programming Language (C++, Java, C#)
- With class, variables and their directly related functions can be grouped together to form a new data type
- It promotes reusability and object-oriented design (not covered in this course)
- Object is an instance of class, i.e. *class* is a blue-print and its product is its *object*.

Class in programming



What is an object?



Class and Object : Example

Without class/object

```
int radius;
int width, height;

double getCircleArea( ) {
    return    3.14*radius*radius;
}
double getRectangleArea() {
    return  width*height;
}
double getCirclePerimeter() {
    return    2*3.14*radius;
}
double getRectanglePerimeter() {
    return  2*(width+height);
}
```

With class/object

```
class Circle {
public:
    int radius;
    double getArea( ) {
        return 3.14*radius*radius;
    }
    double getPerimeter() {
        return 2*3.14*radius;
    }
};
```

```
class Rect {
public:
    int width, height;
    double getArea() {
        return width*height;
    }
    double getPerimeter() {
        return 2*(width+height);
    }
};
```

Class and Object

Without class/object

```
void main(){
    cout << "Please enter the radius of circle";
    cin >> radius;
    cout << getCircleArea();

    cout << "Please enter the width and height of a rectangle";
    cin >> width >> height;
    cout << getRectangleArea();
}
```

With class/object

```
int main() {
    Rect r;   // Rect is a class, r is an object of Rect
    Circle c;
    cout << "Please enter the radius of circle";
    cin >> c.radius;
    cout << c.getArea();
    cout << "Please enter the width and height of a rectangle";
    cin >> r.width >> r.height;
    cout << r.getArea();
    return 0;
}
```


Class in Computer Programming

- An abstract view of real-world objects, e.g. car, horse
- Computer program is a model of real-world problem
- Simple problem: program with variables and functions
- Large scale program: class and object
- Class:
  - definition of program component
  - consists of member variables and member functions
  - Member variable : variable belong to class
  - Member function: function primary designed to access/manipulate the member variable of the class
- Object:
  - An instance of class / runtime representation of a class

Classes and Objects in C++

- A class is a data type, objects are variables of this type
- An object is a variable with member functions and data values
- **cin**, **cou**t are objects defined in header **<iostream>**
- C++ has great facilities for you to define your own class and objects

Defining classes

```
class class_name {
    public / protected / private:
        attribute1 declaration;
        attribute2 declaration;
        method1 declaration;
        method2 prototype;
};
return_value classname::method2 {
        method body statement;
}
```

Defining classes (example I)

```
#include <iostream>
using namespace std;
class DayOfYear {
   public:
      int month;
      int day;
      void output() {
         cout << "month = " << month;
         cout << ", day = " << day << endl;
      }
};
```

Member variables

Member method/function

Member function

- In C++, a class definition commonly contains only the prototypes of its member functions (except for inline functions)
- Use **classname::functionName** to define the member function (method) of particular class.

```
class Circle {
   ...
   int radius;
   ...
   double getArea();
};
double Circle::getArea() {
   return 3.1415*radius*radius;
}
```

Defining classes (example II)

```
#include <iostream>
using namespace std;
class DayOfYear {
public:
   void output(); //member func. Prototype
   int month;
   int day;
};
void DayOfYear::output() {
   cout << "month =" << month << ", day =" << day << endl;
}
```

Define the method elsewhere

Create object, access its function

- To declare an object of a class

  Class_name variable_name;

Examples:

```
Circle c1,c2;
DayofYear today;
```

- A member function of an object is called using the dot operator:

```
today.output();
c1.getArea();
```

Main function

```
void main() {
   DayofYear today, birthday;
    cin >> today.month >> today.day;
    cin >> birthday.month >> birthday.day;
    cout << "Today's date is: ";
    today.output();
    cout << "Your birthday is: ";
    birthday.output();
    if (today.month == birthday.month
              && today.day == birthday.day)
        cout << "Happy Birthday!\n";
}
```


Public and private members

- By default, all members of a class are **private**
- You can declare public members using the keyword **public**
- Private members can be accessed only by member functions (and ***friend*** functions) of that class, i.e. only from within the class, not from outside

A new class definition for DayOfYear

```
class DayOfYear {
Public:
    void input();
    void output();
    void set(int new_m, int new_d);
    int get_month();
    int get_day();
Private:
    bool valid(int m, int d); // check if m,d valid
    int month;
    int day;
};
```


Member function definitions

```
bool DayOfYear::valid(int m, int d) {
    if (m<1 || m>12 || d<1)
        return false;
    switch(m) {
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
        return d<=31; break;
     case 4: case 6: case 9: case 11:
        return d<=30; break;
    case 2:
        return d<=29; break;
    }
}
```

Member function definitions

```
void DayOfYear::input() {
    int m, d;
    // input and validate
    do {
        cout << "Enter month and day as numbers: ";
        cin >> m >> d;  // local var. of input()
    } while (!valid(m,d));
    month = m;  // accessing private members
    day = d;
}
```


Member function definitions

```
void DayOfYear::set(int new_m, int new_d) {
    if (valid(new_m, new_d)) {
        month = new_m;
        day   = mew_d;
    }
}
int DayOfYear::get_month() {
    return month;
}
int DayOfYear::get_day() {
    return day;
}
```

A new main program

```
int main() {
   DayOfYear today, birthday;
   today.input();
   birthday.input();
   cout << "Today's date is:\n";
   today.output();
   cout << "Your birthday is:\n";
   birthday.output();
  if (today.get_month() == birthday.get_month()
                  && today.get_day() == birthday.get_day())
         cout << "Happy Birthday!\n";
  return 0;
}
```


Private Variables and Access functions

- Member functions that give you access to the values of the private member variables are called access functions, e.g., **get_month, set**
- Useful for controlling access to private members:
  - E.g. Provide data validation to ensure data integrity.
- Needed when testing equality of 2 objects. (The predefined equality operator == does not work for objects and variables of structure type.), e.g. **obj1==obj2** (not working!)

Why private variable?

- Prevent others from accessing the variables directly, i.e. variables can be only accessed by access functions.

```
class DayOfYear {
        .........
private:
        int month;
        int day;
        .........
};
```

```
void DayOfYear::set(int new_m, int new_d) {
        ......
    month = new_m;
    day   = mew_d;
        ......
}
int DayOfYear::get_month() {
        return month;
}
int DayOfYear::get_day() {
        return day;
}
```


Why private variables?

- Change of the internal presentation, e.g. variable name, type, will not affect the how the others access the object. Caller still calling the same function with same parameters

```
class DayOfYear {
        .........
private:
        int m;
        int d;
        .........
};
```

```
void DayOfYear::set(int new_m, int new_d) {
        ......
    m = new_m;
    d = mew_d;
        ......
}
int DayOfYear::get_month() {
        return m;
}
int DayOfYear::get_day() {
        return d;
}
```

Why private members?

- The common style of class definitions
  - To have all member variables private
  - Provide enough access functions to get and set the member variables
  - Supporting functions used by the member functions should also be made private
  - Only functions that need to interact with the outside can be made public

Assignment operator for objects

- It is legal to use assignment operator = with objects or with structures
- E.g.

```
DayOfYear due_date, tomorrow;
tomorrow.input();
due_date = tomorrow;
```

- This effectively makes both variables pointing to the same memory address of the object

Constructors for initialization

- Class contains variables and functions
- Variables should be initialized before use in many cases
- In C++, a constructor is designed to initialize variables
- A ***constructor*** is a member function that is automatically called when an object of that class is declared
- Special rules:
  - A constructor must have the same name as the class
  - A constructor definition cannot return a value

Example: Bank account

- E.g., Suppose we want to define a bank account class which has member variables **balance** and **interest_rate**. We want to have a constructor that initializes the member variables.

```
class BankAcc {
public:
        BankAcc(int dollars, int cents, double rate);
  ...
private:
        double balance;
        double interest_rate;
};
...
BankAcc::BankAcc(int dollars, int cents, double rate) {
         balance = dollars + 0.01*cents;
        interest_rate = rate;
}
```

Constructors

- When declaring **BankAcc** objects:

  BankAcc account1(10,50,2.0),
  account2(500,0,4.5);

Note:

A constructor cannot be called in the same way as an ordinary member function is called:

account1.**BankAcc(10,20,1.0);** // illegal

Constructors

- More than one version of constructors are usually defined (overloaded) so that objects can be initialized in more than one way, e.g.

```
class BankAcc  {
   public:
        BankAcc(int dollars, int cents, double rate);
        BankAcc(int dollars, double rate);
        BankAcc();
     ...
   private:
        double balance;
        double interest_rate;
   };
```

Constructors

```
BankAcc::BankAcc(int dollars, int cents, double rate) {
    balance = dollars + 0.01*cents;
    interest_rate = rate;
}
BankAcc::BankAcc(int dollars, double rate) {
    balance = dollars;
    interest_rate = rate;
}
BankAcc::BankAcc() {
    balance = 0;
    interest_rate = 0.0;
}
```


Constructors

- When the constructor has no arguments, don't include any parentheses in the object declaration.
- E.g.

```
BankAcc acc1(100, 50, 2.0), // OK
        acc2(100, 2.3),     // OK
        acc3(),             // error
        acc4;               // correct
```

- The compiler thinks that it is the prototype of a function called **acc3** that takes no argument and returns a value of type **BankAcc**

Constructors

- Alternative way to call a constructor:

  obj = constr_name(arguments);

  E.g., **BankAcc account1;**

  account1 = BankAcc(200, 3.5);

- Mechanism: calling the constructor creates an anonymous object with new values; the object is then assigned to the named object

- A constructor behaves like a function that returns an object of its class type

Default constructor

- A constructor with no parameters
- Will be called when no argument is given

```
class Circle {
public :
        Circle();
        double getArea();
private :
        int radius;
};
void Circle::Circle() {
        radius = 0;
}
double Circle::getArea() {
        return 3.1415*radius;
}
```

```
int main() {
        Circle circle;
        circle.getArea();
        return 0;
}
```

Default constructors

- A default constructor will be generated by compiler automatically if NO constructor is defined.
- However, if any non-default constructor is defined, calling the default constructor will have compilation error.

```
class Circle {
public :
        Circle(int r);
        double getArea();
private :
        int radius;
};
void Circle::Circle() {
        radius = 0;
}
double Circle::getArea() {
        return 3.1415*radius;
}
```

```
void main(){
    Circle circle;      // illegal
    Circle circle(6); //OK
    circle.getArea();
}
```


Copy constructor

```
class Circle {
public :
        Circle();
        Circle(int r);
        Circle(const Circle& c);
        double getArea();
private :
        int radius;
};
Circle::Circle() {   // default constructor
        radius = 0;
}
Circle::Circle(int r) {// constructor
        radius = r;
}
Circle::Circle(const Circle& c) { // copy constructor
        radius = c.radius;
}
double Circle::getArea() {
        return 3.1415*radius;
}
```

```
void main(){
    Circle circle;      // ok
    Circle circle(6); //OK
    Circle newcircle(circle);
    newcircle.getArea();
}
```

Call-by-Reference

- Parameters pass to a function can be updated inside the function
- Add '&' in font of the parameter that to be called by reference
- More detail will be given in further Lecture (Pointer)

```
void swap(int &a, int &b) {
        int temp;
        temp = a;
        a = b;
        b = temp;
}
int main() {
        int x=1,y=3;
        cout << "x:"<<x <<",y:"<<y<<endl;
        swap(x,y);
        cout << "x:"<<x <<",y:"<<y<<endl;
        return 0;
}
```


ALL OF THE REST MATERIALS ARE FOR REFERENCE ONLY.

Friend Function

- Not all functions could logically belong to a class, and sometimes, it is more natural to implement an operation as ordinary (nonmember) functions,
  e.g. Equality (==) function that test if 2 objects are equal
- Equality operator == cannot be applied directly on objects or structures
- Defining it as a member function will lose the symmetry
- It is more natural to define such function as an ordinary (nonmember) function

Equality testing: ordinary function

```
#include <iostream>
using namespace std;
class Rectangle {
public:
    Rectangle(int w,int h);
    int getArea();
    int getWidth();
    int getHeight();
private:
    int width;
    int height;
};
```

```
Rectangle::Rectangle(int w,int h) {
    width = w;
    height = h;
}
int Rectangle::getWidth() {
    return width;
}
int Rectangle::getHeight() {
    return height;
}
int Rectangle::getArea() {
    return width*height;
}
```

Equality testing: ordinary function

```
bool equal(Rectangle r1, Rectangle r2) {
    if (r1.getWidth() == r2.getWidth() && r1.getHeight() == r2.getHeight())
        return true;
    else
        return false;
}
int main() {
    Rectangle ra(10,22), rb(10,21);
    if ( equal(ra, rb) )
        cout << "They are the same\n");
}
```


Friend function

- The previous equality function needs to call access functions several times ⇒ not efficient
- However, declare the member variable as public and direct access them are not recommend

```
class Rectangle {
public:
    int width,height;
……
};

bool equal(Rectangle r1, Rectangle r2) {
if (r1.width == r2.width && r1.height == r2.height)
    return true;
else
    return false;
}
```

Friend function

- Solution: Define a friend function!
- A friend function of a class is not a member function of the class but has access to the private members of that class
- A friend function doesn't need to call access functions → more efficient
- Also the code looks simpler
- A friend function will be public no matter it is defined under "public:" or not

Equality testing: friend function

```
#include <iostream>
using namespace std;
class Rectangle {
public:
    Rectangle(int w,int h);
    friend bool equal(Rectangle r1,Rectangle r2);
    int getArea();
    int getWidth();
    int getHeight();

private:
    int width;
    int height;
};
```

```
Rectangle::Rectangle(int w,int h) {
        width=w;
        height=h;
}
int Rectangle::getWidth() {
        return width;
}
int Rectangle::getHeight() {
        return height;
}
```

Equality testing: friend function

```
/*Note the friend function is not implemented in Rectangle class*/
bool equal(Rectangle r1, Rectangle r2) {
    if (r1.width == r2.width && r1.height == r2.height)
        return true;
    else
        return false;
}

int main() {
    Rectangle ra(10,22), rb(10,21);
    if ( equal(ra, rb) )
        cout << "They are the same\n");
}
```


const modifier revisited

- By default, parameters passed to a function could be call-by-value or call-by-reference mechanism
- Call-by-value: a copy of variable is passed.
- Call-by-reference: the original data, not the copy is passed to a function
- In call-by-reference, if the function is not supposed to change the value of the parameter, you can mark it with a const modifier
- The compiler will then complain when you modify it by mistake

const modifier revisited

- Call-by-reference:
  - the original data, not the copy is passed to a function
  - Add '&' before the parameter name in function prototype and definition.

```
class Rectangle {
    ......
    friend bool equal(Rectangle &r1,Rectangle &r2);
    ......
};
bool equal(Rectangle &r1, Rectangle &r2) {
    if (r1.width ==r2.width && r1.height==r2.height)
    ......
}
```


const parameter modifier

```
class Circle {
private:
    int radius;
public:
    Circle(int r);
    void set(Circle &c);
    double getArea();
};
Circle::Circle(int r) {
    radius=r;
}
void Circle::set(Circle &c) {
    radius = c.radius;
}
double Circle::getArea() {
    return 3.14*radius*radius;
}
```

```
int main() {
    Circle c1(3);
    Circle c2(5);

    cout << c1.getArea();
    cout << '=';
    cout << c2.getArea();
    cout << endl;

    c2.set(c1);

    cout << c1.getArea();
    cout << '=';
    cout << c2.getArea();
    cout << endl;
    return 0;
}
```

const parameter modifier

```
class Circle {
    int radius;
public:
    Circle(int r);
    void set(Circle &c);
    double getArea();
};

Circle::Circle(int r) {
    radius = r;
}
void Circle::set(Circle &c) {
    c.radius = radius;
}
double Circle::getArea() {
    return 3.14*radius*radius;
}
```

```
int main() {
    Circle c1(3);
    Circle c2(5);

    cout << c1.getArea();
    cout << '=';
    cout << c2.getArea();
    cout << endl;

    c2.set(c1);

    cout << c1.getArea();
    cout << '=';
    cout << c2.getArea();
    cout << endl;

}
```


const parameter modifier

```
class Circle {
private:
    int radius;
public:
    Circle(int r);
    void set(const Circle &c);
    double getArea();
};
Circle::Circle(int r){
    radius=r;
}
void Circle::set(const Circle &c){
    c.radius=radius;
}
double Circle::getArea(){
    return 3.14*radius*radius;
}
```

```
void main(){
    Circle c1(3);
    Circle c2(5);

    cout << c1.getArea();
    cout << '=';
    cout << c2.getArea();
    cout << endl;

    c2.set(c1);
    cout << c1.getArea();
    cout << '=';
    cout << c2.getArea();
    cout << endl;
}
```

Compiler will complain!

const modifier for function

- When you have a call to a member, the calling object behaves like a call-by-reference parameter:

```
C1.getArea();
```

- That function may change the value of the calling object

```
double Circle::getArea() {
        return 3.1415*radius*radius++;
}
```

- If you have a member function that is not supposed to change the calling object, you can add the const modifier after the function name (both prototype and definition)

```
double getArea() const;
double Circle::getArea() const {.....
```


const modifier for function

```
class Circle {
private:
    int radius;
public:
    Circle(int r);
    void set(const Circle &c);
    double getArea();
};
Circle::Circle(int r) {
    radius=r;
}
void Circle::set(const Circle &c) {
    radius = c.radius;
}
double Circle::getArea() {
    return 3.14*radius*radius++;
}
```

```
int main() {
    Circle c1(3);
    Circle c2(5);

    cout << c1.getArea();
    cout << '=';
    cout << c2.getArea();
    cout << endl;

    c2.set(c1);

    cout << c1.getArea();
    cout << '=';
    cout << c2.getArea();
    cout << endl;
    return 0;
}
```

const modifier for function

```
class Circle {
private:
    int radius;
public:
    Circle(int r);
    void set(const Circle &c);
    double getArea() const;
};
Circle::Circle(int r){
    radius=r;
}
void Circle::set(const Circle &c) {
    radius = c.radius;
}
double Circle::getArea() const {
    return 3.14*radius*radius++;
}
```

```
void main(){
    Circle c1(3);
    Circle c2(5);

    cout << c1.getArea();
    cout << '=';
    cout << c2.getArea();
    cout << endl;

    c2.set(c1);
    cout << c1.getArea();
    cout << '=';
    cout << c2.getArea();
    cout << endl;
```

Compiler will complain!

const all or nothing

- For each class, use const modifier on an all-or-nothing basis. i.e. All functions called within const function should be a const function too.

```
double Circle::getArea() const {
        return 3.1415*getRadiusSquare();
}
```

- getRadiusSquare() must define as const too, otherwise, compilers will complain as it assumes getRadiusSqaure() will change the value the value of the calling object.

```
int getRadiusSquare() const;

int Circle::getRadiusSquare() const {
        return radius*radius;
}
```

Overloading operators

- An operator is really a function that is called using a different syntax for listing its arguments

- E.g.

  x+y +(x,y) add(x,y)

  x==y ==(x,y) equal(x,y)

- Operators can be overloaded in 2 ways:
  - As a friend function
  - As a member function

Overloading operators: Friend function

```
class Circle {
private:
    int radius;
public:
    Circle(int r);
    void set(const Circle &C);
    double getArea() const;
    int getRadiusSquare() const;
    friend Circle operator+(const Circle &c1,const Circle &c2);
};
```

```
Circle operator+(const Circle &c1,const Circle &c2) {
    Circle c3(c1.radius+c2.radius);
    return c3;
}
int main() {
    Circle c1(3);
    Circle c2(5);
    Circle c3 = c1 + c2;
    cout << c3.getArea();
    return 0;
}
```

Overloading >> and <<

- It is more convenient than using a member function for output

  e.g. **cout << "Area of Circle:" << c1;**

- Equivalent to:

  (cout << "Area of Circle:")<< c1;

- Therefore, the overloaded << operator should return its first argument

Overloading >> and <<

```
class Circle{
    int radius;
public:
    Circle(int r);
    void set(const Circle &C);
    double getArea()  const;
    int getRadiusSquare() const;
    friend Circle operator+(const Circle &c1,const Circle &c2);
    friend ostream &operator << (ostream &outs, const Circle &c);
};
ostream& operator << (ostream& outs, const Circle& c){
    outs << c.getArea();
    return outs;
}
```

Overloading >> and <<

- Whenever an operator (or function) returns a stream, you must add an **&** to the end of the name for the returned type
- Then the operator will return a reference to the stream (instead of the values of the stream)
- Overloading the >> operator:

```
istream& operator >> (istream &ins, Money &amt);
```

- Don't apply **const** modifier to the 2nd parameter

Rules on overloading operators

- When overloading an operator, at least one argument of the resulting overloaded operator must be of a class type
- You cannot create a new operator
- You cannot change the number of arguments that an operator takes
- You cannot change the precedence of an operator

  E.g., **x*y + z** is always interpreted as **(x*y)+z**

Rules on overloading operators

- The following operators cannot be overloaded:

  `. :: .* ?:`

- The following operators can be overloaded but the syntax is different:

  `= [] ->`

Reference Only

- Constructors for automatic type Conversion
- Header (.h) and implementation files (.cpp)
- Namespace
- Structure

Constructors for automatic type conversion

- Consider the example:

```
Circle c1(5),c2(0);
c2= c1 + 25;   //c2 radius =30;
```

- When the system sees the expression:

```
c2= c1 + 25;
```

- it checks if **+** is overloaded for addition between **Circle** and integer.

- If not, it checks if there is a constructor that takes an integer and converts it to **Circle**

Constructors for automatic type conversion

```
class Circle {
    int radius;
public:
    Circle(int r);
    void set(const Circle &C);
    double getArea()  const;
    int getRadiusSquare() const;
    friend Circle operator+(const Circle &c1,const Circle &c2);
};
```

```
int main() {
    Circle c1(3);
    Circle c2(5);
    Circle c3 = c1+5;
    cout << c3.getArea();
    return 0;
}
```

Overloading unary operator

- Similar to overloading binary operators:

```
friend Circle operator -(const Circle &c1, const Circle &c2);
friend Circle operator –(const Circle &c1);
```

- You can overload ++ and -- similarly and use them in the prefix form: **++x --x**

```
friend Circle& operator ++(Circle &c1);
// Don't use const modifier this time!
```

- Overloading for the postfix form is done differently

Separate compilation

- A C++ program can be divided into parts kept in separate files, compiled separately and linked when needed
- Usually, the class definition is placed in a header file (**.h** files)
- The member function definitions are placed in another file (**.cpp** files), which has to include the corresponding .h file
- The main program using the class also needs to include the **.h** file

Header file: circle.h

```
class Circle {
private:
    int radius;
public:
    Circle(int r);
    void set(const Circle &C);
    double getArea()  const;
    int getRadiusSquare() const;
    int getRadius() const;
    friend Circle operator+(const Circle &c1,const Circle &c2);
    friend Circle operator-(const Circle &c1);
    friend ostream &operator << (ostream &outs, const Circle &c);
};
```


Implementation file: circle.cpp

```
#include <iostream>
#include "circle.h"
using namespace std;
Circle::Circle(int r) {
    radius=r;
}

void Circle::set(const  Circle &c) {
    radius=c.radius;
}

double Circle::getArea() const {
    return 3.1415*getRadiusSquare();
}

int Circle::getRadius()const {
    return radius;
}

int Circle::getRadiusSquare() const {
    return radius*radius;
}
```

```
Circle operator+(const Circle &c1,const
    Circle &c2){
    Circle c3(c1.radius+c2.radius);
    return c3;
}

Circle operator-(const Circle &c1) {
    Circle c3(-c1.radius);
    return c3;
}

ostream &operator << (ostream &outs,
    const Circle &c) {
    outs << c.getArea();
    return outs;
}
```

Application file: main.cpp

```
#include <iostream>
#include "circle.h"
using namespace std;

void main(){
    Circle c1(3);
    Circle c2(5);
    Circle c3(0);

    cout << c1.getArea() << '=' << c2.getArea() <<endl;
    c2.set(c1);
    cout << c1.getArea() << '=' << c2.getArea() <<endl;
    c3 = c3 + 1;
    cout << c3;
    c3 = -c3;
    cout << c3.getRadius();
}
```


Separate compilation

- Separate compilation can also be applied to ordinary functions
- The function prototypes of a group of related functions are put in a header file
- Their function definitions are placed in an implementation file (.cpp) which #includes the header file
- The main program is placed in an application file (.cpp) which also #includes the header file

Namespace

- When a program uses different classes and functions written by different programmers, there is a chance of name collision
- A namespace is a collection of name definitions, such as class definitions and variable declarations
- E.g., all name definitions in **<iostream>** (and other standard libraries) are placed in the **namespace std**
- Your program code is placed in the global namespace unless you specify otherwise

Namespace

- If your program (in the global namespace) wants to refer to a name in another namespace, you need to specify that namespace

  E.g., **std::cin**

- If you add the following directive:

  using namespace std;

  then the names in std are added into the global namespace, so you can simply write cin

  Moreover, your program should not use cin as an identifier for other purpose

Namespace

- If a name is defined in two namespaces and your program needs to use them both, you can
  - Use them in different scope:

```
{ using namespace ns1;
  my_function();
}
{ using namespace ns2;
  my_function();
}
```

  - Specify the namespace:

```
ns1::my_function();
ns2::my_function();
```


Structure type & variable definition

- Sometime, we want to have a collection of values of different types and to treat the collection as a single item
- C++ allows users to define structure data types
- Syntax:

```
struct typename {
    type1   member_var1;
    type2   member_var2;
    ......
};
```

```
struct StudentRecord {
    char    name[51];
    char    sid[9];
    float   GPA;
};
```

Structure type & variable definition

- Once a structure type is defined, variables of that type can be defined:

```
struct CS2311Student {
    int  sid;
    float     quiz;
    float     asg1;
    float     asg2;
};
void main(){
    CS2311Student     student;
    cout << "Please enter your id, quiz, a1, and a2 marks\n";
     cin >> sr.id;
    cin >> sr.quiz;
    cin >> sr.asg1;
    cin >> sr.asg2;
    cout << sr.id <<" cw:" <<(sr.quiz+sr.asg1+sr.asg2)/3 << endl;
}
```


Memory allocation & initialization

- When a structure type is defined, no memory is reserved until a variable of that type is declared
- When a variable is declared of a given structure type, enough memory is allocated for storing all structure members contiguously in the main memory
- Initializing a structure variable:

```
CS2311Student danny = {"Danny",50123456,80,75,60};
```

Accessing Individual Members

- A member variable can be accessed with the use of the dot operator "." :

```
danny.quiz += 10;
```

- Two structure types can have the same member name:

```
CS2363Student peter;
cin >> peter.quiz;
```


Assignment for Structures

- You can assign structure values to a structure variable:

```
danny = kitty;
```

which is equivalent to:

```
danny.id   = kitty.id;
danny.quiz  = kitty.quiz;
danny.asg1  = kitty.asg1;
danny.asg2  = kitty.asg2;
```

Structures as Function Arguments

- A function can have parameters of structure type:

```
double overall(CS2311Student s) {
        return s.quiz+ s.asg1+s.asg2)/3;
}
```

- A function can return a value of structure type:

```
CS2331Student InputScore(int);
```


Hierarchical structures

- A member of a structure can be another structure:

```
struct Date{
   int month, day, year;
};
struct PersonInfo{
   double height; //in inches
   double weight; //in pounds
   Date   birthday;
};
PersonInfo peter;
peter.birthday.year=2001;
```