

CS2311 Computer Programming

Class and Object

The need of complex data type (class)

- Real-life programs work with huge amount of information
- Writing functions with only *simple* data type (e.g. `int`, `char`) is troublesome and error-prone...
- Which one is better?
- Example1: Get the number of seconds between two timestamps:
`getTimeDiff(int h1, int m1, int s1, int h2, int m2, int s2);` or
`getTimeDiff(Time t1, Time t2)`
- Example2: Get the distance between two 3D points:
`getDistance(int x1, int y1, int z1, int x2, int y2, int z2);` or
`getDistance(Point p1, Point p2);`
- Obviously those complex data type (`Time`, `Point`...etc) models the problem better and looks more natural.
- This complex data type (*defined by you*) is called **class**

Class and Object

- Class/Object is an important feature of Object-oriented Programming Language (like C++, Java, C#)
- With **class**, variables *and their related functions* can be grouped together to form a new **data type**
- By packaging the data as class, it avoids problems like "forgot to pass param" or "incorrect mix of param (*e.g. x1 and y2*)"
- It simplifies programming and promotes reusability
- **Object** is an instance of a class
(*i.e. class is a blue-print and object is the product.*)
- `int x;` (`int` is "**data type**", `x` is "**variable**")
 `Point p1;` (`Point` is "**class**", `p1` is "**object**")

Class and Object : Example

Without class/object

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

With class/object

```
class Circle {  
    public:  
    int radius;  
    double getCircleArea(){  
        return 3.14*radius*radius;  
    }  
    double getCirclePerimeter(){  
        return 2*3.14*radius;  
    }  
};  
  
class Rect{  
    public:  
    int width, height;  
    double getRectangleArea(){  
        return width*height;  
    }  
    double getRectanglePerimeter(){  
        return 2*(width+height);  
    }  
};
```

Don't forget ;

Class and 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();  
}
```

Without class/object

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

With class/object

Class in Computer Programming

- An abstract view of real-world objects.
- Class:
 - Commonly consist of **member variables** and **member functions**
 - **Attribute** (*also known as property / member variable*):
Variable belong to class
 - **Method** (*also known as "member function"*):
Function primary for accessing the member variable of the class
- Object:
 - Instance of class / runtime representation of a class

Class as a *blue-print*

```
void start();  
void shutdown();  
void moveForward(int step);  
void turnLeft(int degree);  
void turnRight(int degree);  
void takePhoto();  
.....
```

*... as well as the
supported operations...*

Class:Robot



Member
variable

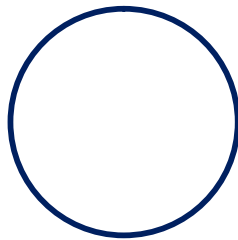
Member
functions

*You design the
internal data...*

```
int modelNum;  
int width;  
int height;  
int powerLevel;  
.....
```

Object as a *product* following the blue-print

Class: Circle

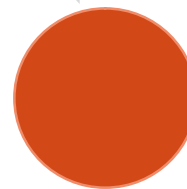


int radius;
int color;

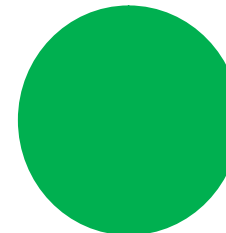
Member
functions

Objects of Circle

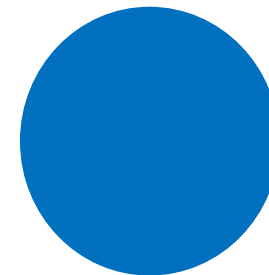
Radius:10
Color:orange



Radius:12
Color: green



Radius:15
Color: blue



Classes and Objects in C++

- A class is a data type whose variables are objects
- An object is a variable with member functions and data values
- Examples of classes provided by C++:
 - `ifstream`, `ofstream` are classes for file read/write, defined in header `<fstream>`
 - `cin`, `cout` are objects defined in header `<iostream>`
- C++ also allows you to *define your own class* and objects

Defining classes

```
class class_name {  
    public / protected / private:  
    attribute1 declaration;  
    attribute2 declaration;  
    method1 declaration;  
    method2 prototype;  
}; ← Don't forget ;  
  
    return_value classname::method2{  
        method body statement;  
    }
```

Defining classes (example I)

You can place the output() function directly in class

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

Member variables

Member function

Defining classes (example II)

Alternatively, you may also place the output() function outside class. *(but requires the **DayOfYear::** prefix)*

```
#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;
}
```

Function body
placed outside class

Main function

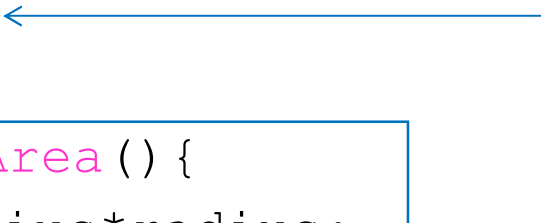
```
void main()
{
    DayofYear today, birthday;
    cin >> today.month >> today.day;
    cin >> birthday.month >> birthday.day;
    cout << "Today's date is: ";
    today.output();
    cout << "Your birthday is: ";
    birthday.output();
    if (today.month == birthday.month
        && today.day ==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

Member function

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

```
class Circle
{
    .....
    int radius;
    .....
    double getArea();
};

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



Create object and access its member function

- To create an object of a class

Class_name object_name;

Examples:

```
Circle c1, c2;
```

```
DayofYear today;
```

- A member function / variable of an object is accessed using the **dot operator**:

```
• cout << today.month;           //member variable
```

```
• today.output();                //member function
```

```
• c1.getArea();
```

Public and **private** members

- By **default**, all members of a class are **private**
- You can declare public members using the label **public**
- **Private members** can be accessed only by member functions (and *friend* functions) of that class, so as to avoid problems like:

```
DayOfYear today;  
    today.month = 13;
```

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

```
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 (cont'd)

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

int DayOfYear::get_month()
{ return month;
}

int DayOfYear::get_day()
{ return day;
}
```

month and *day* are
guaranteed to be correct
because they are *private*

The only way to change them
is to call *set()*, which
performs validation

Member function definitions (cont'd)

```
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;
    }
}
```

Actually the **break;**
here is optional

A new main program

```
void 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";
}
```

Private Variable 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()`
(Some may further classify access functions as *getter* and *setter*)
- Useful for controlling access to private members:
 - E.g. Provide data validation to ensure data integrity.
- Just like the program in the last page, we used `get_day` and `get_month` to check the equality of two objects.
(Because the `==` operator does not work for objects/structures!)
e.g. `today==birthday` does not work!

Why private variable?

- Prevent others access the variable directly, i.e. variable can be only accessed by access function.

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

    .....
};
```

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

int DayOfYear::get_month()
{
    return month;
}

int DayOfYear::get_day()
{
    return day;
}
```

Why private variable?

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

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

Renamed **month** / **day** to **m** / **d**, but also works!

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

int DayOfYear::get_month()
{
    return m;
}

int DayOfYear::get_day()
{
    return d;
}
```

Usual 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 (*e.g. the `valid()` function of our `DayOfYear` class, which supports `input()` and `set()`)*

Assignment operator for objects

- To copy data among objects, you may simply use the `=` operator.
- E.g.

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

Constructors for initialization

- Variables must be initialized before use.
For simple variables (*like integer*), it is easy: `int a=5;`
- However, an object contains *multiple* variables and functions
`DayOfYear today=?????;`
How can we set `month` and `day` at the same time?
- In C++, constructor is designed to initialize variables
- A *constructor* is *like* a member function which is automatically called when an object of that class is created
- Special rules:
 - A constructor must have the *same name as the class*
 - A constructor definition *has NO return a value (not even void!)*

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;
}
```

Prototype of constructor

Body of constructor

Constructors

- When declaring objects of BankAcc class:

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

- Note: A constructor cannot be called explicitly as it is NOT really a function! :

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

Constructors

- More than one versions 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;
}
```

default constructor
Automatically called when
you write

BankAcc acc4;

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 arguments and returns a value of type `BankAcc`

Constructors

- Alternative way to make use of constructor:

obj = constr_name(arguments);

E.g., BankAcc account1;
 account1 = BankAcc(200, 3.5);

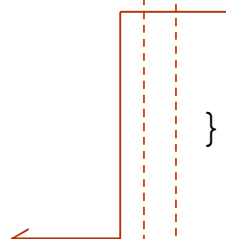
- Mechanism: calling the constructor creates an anonymous object with new values; the object is then assigned to the named object

Default constructors

- A constructor with no input parameter
- Will be called when no argument is given (e.g. *Circle c;*)
- If you do not redefine it, by default, it does nothing at all!

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

```
void main() {
    Circle c;
    c.getArea();
}
```



Default constructors

- Default constructor will be **generated** by compiler automatically **if there is NO other constructor**.
- However, if any non-default constructor is defined, the compiler will not generate it for you. As such, using the default constructor will generate compilation error.

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

```
void main(){
    Circle c; //illegal: No default!
    Circle c(6); //OK
    c.getArea();
}
```

friend function

- Sometimes, it is more natural to implement an operation as an ordinary (non-member) function:
- As mentioned, equality operator `==` **cannot** be applied directly to objects. So if we'd like to compare objects `obj1` and `obj2`, we need to write our own functions.

- Which one would you prefer ?

```
if (equals(obj1,obj2)) ...      or  
if (obj1.equals(obj2)) ...      ?
```

- The second method, (i.e. `obj1.equals(obj2)`), is not natural as it loose symmetry (why not `obj2.equals(obj1)`?)

Equality testing: *without* friend 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: *without* friend function

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

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

friend function

- Defining `equal()` as non-member function looks natural, however, we need to call access functions `getWidth()` and `getHeight()` multiple times $\Rightarrow$ **not efficient!**
- However, declaring the member public is not desirable as well (*because of consistency problem*)

```
• 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 does not need to call the accessor functions to access the private members $\Rightarrow$ more efficient!
- Also, the code looks simpler

Equality testing: *WITH* friend function

```
#include <iostream>
using namespace std;
class Rectangle
{
public:
    Rectangle(int w,int h);
    friend 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: *WITH* friend function

*/*Note the friend function is NOT implemented inside the Rectangle class*/*

← **NO NEED** to write **friend** here, otherwise syntax error!

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

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

Check-list

After this lecture, students are expected to be able to:

- Explain what is, and why we need classes
- Create classes with public and private attributes and methods
- Define function (*method*) body inside or outside the class definition.
- Explain the pros and cons of public / private
- Prepare two or more constructors for class initialization
- Explain the differences of syntax between function-based programs against class-based programs.