CS2311 Computer Programming

LT11: Class & Object

Supplementary

For Reference Only

Constructors

```
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*radius;
}
```

```
int main() {
  Circle circle;       // ok
  Circle circle(6); // ok

  Circle newcircle(circle);
  cout << newcircle.getArea();
  return 0;
}
```

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;
};
```

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

`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 Parameter Modifier

```
class Circle {
public:
   Circle(int r);
   void set(const Circle& c);
   double getArea();
private:
   int radius;
};
Circle::Circle(int r){
   radius=r;
}
void Circle::set(const 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;
   return 0;
```

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++;
}
```

```
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() << endl;
   return 0;
}
```

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 member function
  - As a friend function

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`

Overloading Operators: Member function

```
class Circle {
private:
   int radius;
public:
   Circle(int r);
   double getArea() const;
   Circle operator +(const Circle &c) const;
   bool operator <(const Circle& c) const;
};
```

```
int main() {
   Circle c1(3);
   Circle c2(5);
   Circle c3 = c1 + c2;
   cout << c3.getArea();
   if (c1 < c2)
      cout << "c1 < C2");
   else
      cout << "c1 >= c2";
   return 0;
}
```

```
Circle Circle::operator+(const Circle &c) const {
   return Circle(radius+c.radius);
}
bool Circle::operator<(const Circle &c) const {
   return (radius < c.radius);
}
```

Overloading unary Operator

- Similar to overloading binary operators:

```
Circle operator -() const;
```

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

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

- Overloading for the **postfix** form is done differently

```
Circle operator ++(int);
```

Rules on Overloading Operators

- The following operators cannot be overloaded:

  `.` `::` `*`(pointer) `?:`

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

  `=` `[]` `->`

`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

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;
   Circle operator+(const Circle &c1) const;
   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

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();
  Circle(int r);
  double getArea()  const;
  int getRadiusSquare() const;
  Circle operator +(const Circle &c) const;
  Circle operator -(const Circle &c) const;
  friend ostream &operator <<(ostream &outs, const Circle& c);
  friend istream& operator >>(istream& ins, Circle& c);
};
```

Implementation file: circle.cpp

```
#include <iostream>
#include "circle.h"
using namespace std;
Circle::Circle() {
   radius = 0;
}
Circle::Circle(int r) {
   radius = r;
}
double Circle::getArea() const {
   return 3.1415*getRadiusSquare();
}
int Circle::getRadiusSquare() const {
   return radius*radius;
}
```

```
Circle Circle::operator+(const
   Circle &c) const {
   return Circle(radius+c2.radius);
}
Circle Circle::operator-() const {
   return Circle(-radius);
}
ostream& operator <<(ostream& outs,
   const Circle &c) {
   outs << c.getArea();
   return outs;
}
istream& operator >>(istream& ins,
   Circle& c) {
   ins >> c.radius;
   return ins;
}
```

Application file: main.cpp

```
#include <iostream>
#include "circle.h"
using namespace std;
int main(){
  Circle c1(3);
  Circle c2(5);
  Circle c3;
  cout << c1 << " " << c2 << endl;
  c3 = c3 + 1;
  cout << c3;
  c3 = -c3;
  cout << c3 « endl;
  return 0;
}
```

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