

CS2311 Computer Programming

Reference : Class and Object

for interested students

Table of Content

- *const* modifier
- Operator overloading
- Constructor for automatic type conversion
- Separate file compilation
- *namespace*
- structures

const modifier revisited

- Parameters passed to a function could be done via the pass-by-value or pass-by-reference mechanisms
- **Pass-by-value**: a **copy** of variable is passed.
- **Pass-by-reference**: the **original data**, not the copy is passed
- One common reason to use pass-by-reference is that we want to **modify the original** inside function
- However, there is another reason: **Runtime efficiency!**
- Obviously an object is much larger than an integer (*in the member variables, there may be another object / array !*)
- Using pass-by-value will make a copy of the whole object which is highly inefficient!

const modifier revisited

- However, if we pass objects using pass-by-reference, there's a risk that we may accidentally changed the members
(despite we don't really want to, we just want performance!)
- In pass-by-reference, if the function is not supposed to change the value of the parameter, you can mark *each* parameter with a `const` modifier
- The compiler will then complain when your function changes the parameter by mistake

Usage 1 – `const` modifier with input parameters

- Call-by-reference:
 - the original data, not the copy is passed to a function
 - Add '&' before the parameter in function prototype and in the definition of function body.
 - Program is more efficient ! (*possibly faster*)

```
class Rectangle
{
    .....
    friend equal(Rectangle &r1, Rectangle &r2);
    .....
};

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

Usage 1 – `const` modifier with input parameters

- Since we *just* want to read (*but not update*) the rectangles, we put the `const` keyword before the parameters.
 - Make the input parameters read-only
 - Protect from accidental writes

```
class Rectangle
{
    .....
    friend equal(const Rectangle &r1, const Rectangle &r2);
    .....
};

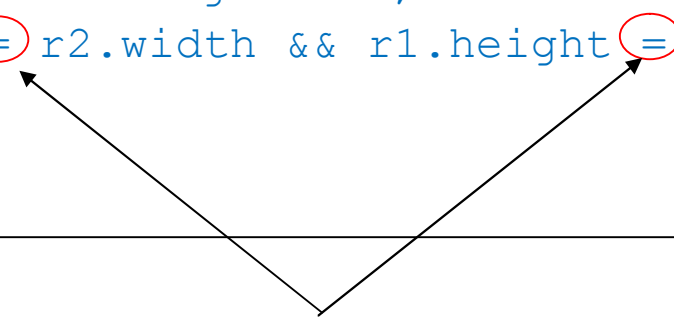
bool equal(const Rectangle &r1, const Rectangle &r2){
    if (r1.width == r2.width && r1.height == r2.height)
        .....
}
```

Usage 1 – `const` modifier with input parameters

- If we tried to (or accidentally) update the members in Rectangles, compiler will result in compilation error.

```
class Rectangle
{
    .....
    friend equal(const Rectangle &r1, const Rectangle &r2);
    .....
};

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



If we mistakenly wrote `==` as `=`. Compiler will complain for that because we are not supposed to update `r1.width` and `r1.height`

Usage 2 – `const` modifier with read-only function

(not related to input parameters!)

- Member functions could access *(and modify!)* the private members of an object. However we may not always want to do that!

- Some functions are just supposed to **report** the result and therefore should read-only. For example:

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

- To declare those read-only functions, you can add the `const` modifier after the function name *(both prototype and definition)*

```
double getArea() const;
```

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

Usage 2 – **const** modifier with read-only function

(not related to input parameters!)

```
class Circle
{
    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;
}
```

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

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

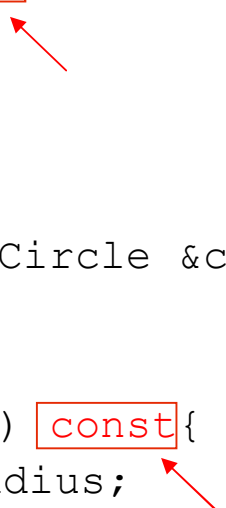
    c2.set(c1);

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

Usage 2 – **const** modifier with read-only function

(not related to input parameters!)

```
class Circle
{
    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;
}
```

Usage 2 – `const` modifier with read-only function

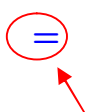
(not related to input parameters!)

- Suppose we change the implementation:

```
double Circle::getArea() const{  
    return (radius==0) ? 0 : 3.14*radius*radius;  
}
```

- And then we made a typo (`==` typed as `=`):

```
double Circle::getArea() const{  
    return (radius=0) ? 0 : 3.14*radius*radius;  
}
```



- Compiler will issue compilation error as we are not suppose to change the `radius` member of the object.

Usage 2 – `const` modifier with read-only function

(not related to input parameters!)

- Note: All functions called by a `const` function should also be `const`

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

- `getRadiusSquare()` must be defined 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;  
}
```

Difference between Usage 1 and Usage 2

```
class Circle
{
    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;
}
```

Usage1:

- Used only for pass-by-reference
- To protect input parameter
- Can be used for other non-class functions.

Usage2:

- Used only for class functions
- To protect class variables
- May be used together with **Usage1**

Overloading operators

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

- For example:

$x+y$
 $x==y$ is actually $\text{operator } +(x,y)$
 $\text{operator } ==(x,y)$ which is similar to $\text{add}(x,y)$
 $\text{equal}(x,y)$

- Operators can be overloaded in 2 ways:

As a friend function (more flexible, object in any order)

As a member function (restriction: 1st operand must be object)

Overloading operators: Friend function

```
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);
};
```

Return type:
c1+c2 is also
a Circle!

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

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

Pass-by-reference:
Better performance

const:
Avoid accidental change

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 (e.g. @)
- You cannot change the number of arguments that an operator takes (e.g. create a % which takes 1 number)
- 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: *(not covered in this course)*

= [] ->

Overloading unary operator

- Similar to overloading binary operators:

minus → friend Circle operator -(const Circle &c1,
const Circle &c2);
negate → 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

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 (*i.e. cout in this case*)

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 returned type
- Then the operator will return a reference to the stream (i.e. the original copy)
- Overloading the >> operator:

```
istream & operator >>(istream &ins, Money &amt);
```
- Don't apply `const` modifier to the 2nd parameter

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`. Finally it calls the `+` operator between `Circle` and `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);
};
```

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

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  sr;  
    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 = {50123456, 80, 75, 60};
```

- Note that it is different from class, which uses constructor !

Accessing individual members

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

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

```
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;
```

- Meanwhile for class/object, the exact behavior depends on the definition of the `=` operator (if any)

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

- 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;
```

- Similarly, member variable / attribute of an object could be another object