10. Classes II

15. Fine Tuning Objects
16. Building Some Useful Classes

Passing Object Parameters

```
class A {
public:
    int x;
    A(int x) : x(x) {}
    void printX() {
        std::cout << x << std::endl;
    }
};
void Print1(A a) {
    a.printX();
}
void Print2(const A& a) {
    std::cout << a.x << std::endl;
    // a.printX();
}
int main() {
    A a1(1);
    a1.printX();
    Print1(a1);
    Print2(a1);
}
```

reference: passing an instance/no need to copy an object
const: cannot modify

const Methods (1)

```
class A {
    int x;
public:
    A(int x) : x(x) {}
    void printX() const {
        std::cout << x << std::endl;
    }
};
void Print(const A& a) {
    a.printX();
}
int main() {
    A a1(1);
    a1.printX();
    Print(a1);
}
```

Methods declared to be const can be called with const objects, while it is illegal to invoke a non-const method with a const object.

const Methods (2)

```
#include <iostream>
class A {
    int x, y;
public:
    A(int x, int y) : x(x), y(y) {}
    void printX() {
        std::cout << "non-const " << x << std::endl;
    }
    void printX() const {
        std::cout << "const " << x << std::endl;
    }
    void printY() const {
        std::cout << y << std::endl;
    }
};
int main() {
    A a1(1, 2);
    a1.printX();            a1.printY();

    const A a2(3, 4);
    a2.printX();            a2.printY();
}
```

Pointers to Objects and Object Arrays (1)

```
Account acct("Joe", 3143, 90.00);
Account *acct_ptr;

acct_ptr = &acct;
// acct_ptr = new Account("Moe", 400, 1300.00);
// delete acct_ptr; // [] ?

(*acct_ptr).id = 100;
acct_ptr->id = 100;    // arrow member selection operator(->)
```


Pointers to Objects and Object Arrays (2)

```
std::vector<Point> pts(100);     // Okay
std::vector<Account> accts(100); // Illegal, the Account class has
                                 // no default constructor

std::vector<Account *> pAccts(100); // A vector of account pointers
for (int i = 0; i < 100; i++) {
   std::cin >> name >> id >> amount;
   pAccts[i] = new Account(name, id, amount);
}
for (int i = 0; i < 100; i++) {
   delete pAccts[i];

Account *p_rec = nullptr;
// A pointer with the value nullptr is interpreted to mean a pointer
// that is pointing to nothing. An attempt to delete a null pointer
// is legal and does nothing.
```

Pointers to Objects and Object Arrays (3)

```
std::vector<Account> accts; // Vector initially empty
for (int i = 0; i < 100; i++) {
   std::string name;          int id;        double amount;
   std::cin >> name >> id >> amount;
   accts.push_back({name, id, amount});
}
```


The this Pointer (1)

```
class Counter {// Within a method body the this expression
   int count;  // represents a pointer to the object upon which the
public:        // method was called.
   void clear() {
      count = 0;
   }
   void inc() {
      this->count++;
   }
};

class Point {
   double x; double y;
public:
   void set_x(double x) {
      this->x = x;
   }
};
```

The this Pointer (2)

```
class Point {
    int x, y;
public:
    Point(int x, int y) : x(x), y(y) {}
    Point* Set(int x, int y) {
        *this = { x, y };
        return this;
    }
    void Inc() {
        x++;
        y++;
    }
    void Print() const {
        std::cout << x << ", " << y << std::endl;
    }
};
```


Separating Method Declarations/Definitions

```
// Point.h
class Point {
   double x;
   double y;
Public:
   Point(double x, double y); // No constructor implementation
   double get_x() const;      // and no method
   double get_y() const;      // implementations
};

// Point.cpp
Point::Point(double x, double y): x(x), y(y) {}
double Point::get_x() const {
   return x;
}
double Point::get_y() const {
   return y;
}
```

inline method definition
Compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime.

Preventing Multiple Inclusion (1)

```
class O {
};
```

```
#include "O.h"
class A{
};
```

```
#include "O.h"
class B{
};
```

```
#include "A.h"
#include "B.h"
int main() {

}
// class type redefinition-error
```


Preventing Multiple Inclusion (2)

```
#ifndef O_H_
#define O_H_
class O {
};
#endif
```

```
#include "O.h"
class A{
};
```

```
#include "O.h"
class B{
};
```

```
#include "A.h"
#include "B.h"
int main() {

}
// class type redefinition-error
```

Predefined Macros

```
__FILE__, __LINE__


_MSC_VER
__linux__
__ANDROID__
```


Overloaded Operators (1)

Operator	Class	Function	Example	Chapter
`operator<<`	`std::ostream`	Right-hand operand is sent to the stream specified by the left-hand operand	`std::cout << x;`	Chapter 2
`operator>>`	`std::istream`	Right-hand operand is extracted from the stream specified by the left-hand operand	`std::cin >> x;`	Chapter 4
`operator[]`	`std::vector`	Right-hand operand (the integer within the square brackets) is used to locate an element within the left-hand operand (a `vector`)	`x = v[2];`	Section 11.1
`operator==`	`std::string`	Right-hand operand (a `string`) is compared with the left-hand operator (also a `string`) to determine if they contain exactly the same characters in exactly the same order	`if (word == "Please") proceed();`	Chapter 13

Overloaded Operators (2)

```
class Point {
public:
   double x;
   double y;
};

Point operator+(const Point& p1, const Point& p2) {
   Point result;
   result.x = p1.x + p2.x;
   result.y = p1.y + p2.y;
   return result;
}

std::ostream& operator<<(std::ostream& os, const Point& pt) {
   os << '(' << pt.get_x() << ',' << pt.get_y() << ')';
   return os;
}
```


Overloaded Operators (3)

```
class Point {
public:
   double x;
   double y;

   Point operator+(const Point& p) const;
};

Point Point::operator+(const Point& p) const {
   Point result;
   result.x = x + p.x;
   result.y = y + p.y;

   return result;
}
```

static Members

```
// By default, a class member is an instance member.
#include <iostream>   // Static members are instance independent.
#include <cmath>
class Point {
public:
   double x;   double y;
   static double pi;
   static double Distance(double x1, double y1, double x2, double y2);
};
double Point::pi = 3.14159;
double Point::Distance(double x1, double y1, double x2, double y2) {
   return sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
}
int main() {
   Point p1, p2;       double r = 5;
   p1.x = 10;  p1.y = 5;      p2.x = 15;     p2.y = 25;
   std::cout << Point::Distance(p1.x, p1.y, p2.x, p2.y) << std::endl;
   std::cout << Point::pi*r*r << std::endl;
}
```


Classes vs. Structures

```
// In a class, all members are private by default.
// In a structure, all members are public by default.

struct Point {
   double x; // These fields now are public
   double y;
};
```

Friends (1)

```
class ReadOnlyRational {
   int numerator;
   int denominator;
public:
…
   int get_numerator() const {
      return numerator;
   }
   int get_denominator() const {
      return denominator;
   }
};
std::ostream& operator<<(std::ostream& os, const ReadOnlyRational& f)
{
   os << f.get_numerator() << '/' << f.get_denominator();
   return os;
}
```


Friends (2)

```
// Friend functions are not members of a class, but are associated
// with it. They can access the private members of the class as
// though they were members.
// Friend classes can access the private members of the class which
// is declaring

#include <iostream>
#include <cstdlib>
class PrintOnlyRational {
   int numerator;
   int denominator;
public:
   PrintOnlyRational(int n, int d): numerator(n), denominator(d) {
      if (d == 0) {
         std::cout << "Zero denominator error\n";
         exit(1);
      }
   }
```

Friends (3)

```
   PrintOnlyRational(): PrintOnlyRational(0, 1) {}
   friend std::ostream& operator<<(std::ostream& os,
      const PrintOnlyRational& f);
};
std::ostream& operator<<(std::ostream& os, const PrintOnlyRational& f)
{
   os << f.numerator << '/' << f.denominator;
   return os;
}
int main() {
   PrintOnlyRational fract{1, 2}; // The fraction 1/2
   std::cout << "The fraction is " << fract << '\n';
   PrintOnlyRational fract2{2, 3}; // The fraction 2/3
   std::cout << "The fraction is " << fract2 << '\n';
}
```


Friends (4)

```
#include <iostream>
class Widget {
   int data;
public:
   Widget(int d): data(d) {}
   friend class Gadget;
};
class Gadget {
   int value;
public:
   Gadget(const Widget& w): value(w.data) {}
   int get() const {   return value;   }
   bool compare(const Widget& w) const {   return value == w.data;   }
};
int main() {
   Widget wid{45};   Gadget gad{wid};
   std::cout << gad.get() << '\n';
   if (gad.compare(wid))
      std::cout << "They are the same" << '\n';
}
```

A Better Rational Number Class (1)

```
#include <iostream>
class Rational {
   int numerator;
   int denominator;
   // Compute the greatest common divisor (GCD) of two integers
   static int gcd(int m, int n) {
      if (n == 0)
         return m;
      else
         return gcd(n, m % n);
   }
   // Compute the least common multiple (LCM) of two integers
   static int lcm(int m, int n) {
      return m * n / gcd(m, n);
   }
```


A Better Rational Number Class (2)

```
public:
   Rational(int n, int d): numerator(n), denominator(d) {
      if (d == 0) {
         std::cout << "*****Warning---Illegal Rational\n";
         numerator = 0; // Make up a reasonable default fraction
         denominator = 1;
      }
   }
   // Default fraction is 0/1
   Rational(): numerator(0), denominator(1) {}

   int get_numerator() const {
      return numerator;
   }
   int get_denominator() const {
      return denominator;
   }
```

A Better Rational Number Class (3)

```
   Rational reduce() const {
      int factor = gcd(numerator, denominator);
      //return Rational(numerator/factor, denominator/factor);
      return {numerator/factor, denominator/factor};
   }
   // Equal fractions have identical numerators and denominators
   bool operator==(const Rational& fract) const {
      Rational f1 = reduce(),
      f2 = fract.reduce();
      // ...then see if their components match.
      return (f1.numerator == f2.numerator)
         && (f1.denominator == f2.denominator);
   }
```


A Better Rational Number Class (4)

```
   bool operator!=(const Rational& other) const {
      return !(*this == other);
   }
   Rational operator+(const Rational& fract) const {
      int commonDenominator = lcm(denominator, fract.denominator);
      int newNumerator = numerator * commonDenominator/denominator
         + fract.numerator * commonDenominator/fract.denominator;
      return {newNumerator, commonDenominator};
   }

   Rational operator*(const Rational& fract) const {
      return Rational(numerator * fract.numerator,
         denominator * fract.denominator).reduce();
   }
};
```

A Better Rational Number Class (5)

```
std::ostream& operator<<(std::ostream& os, const Rational& r) {
   os << r.get_numerator() << "/" << r.get_denominator();
   return os;
}

int main() {
   Rational f1(1, 2), f2(1, 3);
   std::cout << f1 << " + " << f2 << " = " << (f1 + f2) << '\n';
   std::cout << f1 << " * " << f2 << " = " << (f1 * f2) << '\n';
}
```