by Leedehai

Chapter 11 Working with Classes

11.1 Operator overloading

- *Operator overloading* is a grammar candy that enables you to add additional definitions to an existing operator when applied to operand(s) of a class type. Judicious use of this technique can make programs easier to write and read.
- Operator overloading is implemented by *operator function*s.
  - Operator functions have special names: the keyword `operator` followed by the symbol for the operator being overloaded.
  - An operator function has a return type, a argument list and a definition body, like any other functions.
  - An operator function can be implemented as a non-member or member functions, with an exception that `=`, `->`, `()`, `[]` can only be overloaded as member functions.
  - Two equivalent calling syntaxes: normal-like, function-like. i.e.

```
// non-member operator function
vector1 = vector1 + vector2;   // normal-like
operator+(data1,data2);        // function-like
// member operator function
tensor1+=tensor2;              // normal-like
tensor1.operator+(tensor2);    // function-like
```

- Restrictions for operator overloading
  - The operator must be a valid C++ operator. You can create your own.
  - The operator, when overloaded, must NOT be used in a manner that violets its original syntax rules. That is, (1) you cannot alter its operator precedence and associativity; (2) you cannot change the number of operands it takes (unary, binary).

    > Four symbols (`+`, `-`, `*`, `&`) serve as both unary and binary operators. Either or both usages of an operator of these can be overloaded. The number of arguments indicates the overloaded operator is unary or binary.

  - At least one operand for a overloaded operator must be of user-defined type, be it a structure type or a class type.
  - If the operator function is a member function of a class, the invoking operand must be of the class type (for binary operators, the invoking operand is the left one). The invoking operand is passed to the function implicitly (not in the argument list).
  - The operator's meaning stays unchanged when applied to operands of built-in types.
  - You can overload most, but not all, operators.
    - Operators that CAN be overloaded by using either member or non-member functions:

      arithmetic: `+` `-` `*` `/` `%` bitwise: `^` `&` `|` `~` bit shift: `>>` `<<`

      combination assignment: `+=` `-=` `*=` `/=` `%=` `^=` `&=` `|=` `>>=` `<<=`

relational: `<` `>` `>=` `<=` `==` `!=` logic: `&&` `||` `!`

self-increment/decrement: `++` `--`

others: `->*` `,` `new` `new[]` `delete` `delete[]`

- Operators that CAN be overloaded only by using member functions:

  `=` (assignment) `()` (function-call) `[]` (subscripting)

  `->` (member-access-by-pointer) `type()` (type-conversion)

- Operators that CANNOT be overloaded:

  `sizeof` `::` `?:` `.` (membership) `.*` (pointer-to-member) `typeid` (RTTI)

  `static_cast` `dynamic_cast` `const_cast` `reinterpret_cast`

- Choosing member or non-member implementation
  - These binary operators must be defined as members: `=`, `[]`, `()`, `->`.
  - Type-conversion operators must be defined as members: `int()`, `double()`, ...

    > The "type" in a `type()` operator can be ANY type that can be a function return type but `void`, be it a built-in type, structure type, or a class type.

  - Operators that change the state of their object, or that are closely ties to their given type, are recommended to be members.
  - Symmetric operators - those might convert either operand, such as the arithmetic, equality, relational, and bitwise operators - are recommended to be non-member. An example:

```
string s = "world"; string t = s + "!"; stirng u = "hi" + s; // ok
// the last one would be an error if '+' were a member of 'string'
// class, because "hi".operator+(s) is invalid, where "hi" is
// 'const char *', not a 'string' object.
```

- Some simple examples
  - Operators as member functions

```
bool Stock::operator>(Stock &st){           // a binary operator '>'
  return price > st.price ? true : false;   // compare prices
}  // usage: if(stocks[0]>stocks[1]) {...}
   // or:    if(stocks[0].operator>(stocks[1])) {...}
```

```
void Stock::operator!(){                     // a unary operator '!'
  shares = 0;  // liquidate the stock
}  // usage: !stocks[2];  or:  stocks[2].operator!();
```

```
double Stock::operator[](std::string s){   // a binary operator '[]'
  if(s=="open") return opnPrice; if(s=="closing") return clsPrice;
  if(s=="max")  return maxPrice; if(s=="min")     return minPrice;
}  // usage: double a = stocks[0]["max"];
   // or:    double a = stocks[0].operator[]("max");
```

```cpp
void Stock::operator*(unsigned float f){   // a binary operator '*'
  shares = shares * f;
}  // usage: stocks[1]*1.25;  or:  stocks[1].operator*(1.25);
```

- Operators as non-member functions

```cpp
ostream &operator<<(ostream &os, const Stock &stock){ // binary '<<'
  os << "COMPANY: " << stock.company << "\t PRICE: " << stock.price
    << "\t SHARES: " << stock.shares << std::endl;
  return os; // so consecutive output is supported: cout << .. << ..;
}   // usage: std::cout << "stock 0:" << stocks[0] << std::endl;
    // or:  operator<<((cout << "stock 0:"), stocks[0]) << std::endl;
// NOTE that this function has to be a friend to the class "Stock",
// so as to access Stock objects' private and protected members.
```

```cpp
istream &operator>>(istream &is, Stock &stock){     // binary '>>'
  is >> stock.company >> stock.price >> stock.shares;
  if (!is)// if input fails, give the object the default state
    stock = Stock();
  else stock.total_val = stock.set_total();
  return is; // so consecutive input is supported: cin >> .. >> ..;
}   // usage: std::cin >> stocks[2]; (type in: WeissArt 5.6 200)
    // or: operator>>(std::cin,stocks[2]);
// NOTE that this function has to be a friend to the class "Stock",
// so as to access Stock objects' private and protected members.
```

```cpp
bool operator>(Stock &st1, Stock &st2){             // binary '=='
  return st1.price > st2.price ? true : false;  // compare prices
}  // usage: if(stocks[0]>stocks[1]) {...}
   // or:    if(operator>(stocks[0],stocks[1])) {...}
// NOTE that this function has to be a friend to the class "Stock",
// so as to access Stock objects' private and protected members.
```

```cpp
void operator*(unsigned float f, Stock stock){stock*f;} // binary '*'
   // usage: 1.25 * stocks[1];  or:  operator*(1.25, stocks[1]);
// This function does NOT have to be a friend to the class "Stock",
// since it use Stock objects as a whole, as it does not use members.
```

- Special case: overloading prefix and postfix `++` / `--`. The operator functions can be ether member or non-member.
  - For prefix `++` / `--`, the functions are as usual;
  - For postfix `++` / `--`, the function has an extra argument of type `int`. When calling the operator function, you should pass an arbitrary value to this extra argument.

```cpp
StockPtr &StockPtr::operator++();     // prefix, member function
StockPtr &StockPtr::operator++(int);  // postfix, member function
```

```
FuturePtr operator++(FuturePtr);      // prefix, non-member function
FuturePtr operator++(FuturePtr, int); // postfix, non-member function
```

- Use operator overloading judiciously
  - Each operator has an associated meaning from its use on the built-in types.
  - Operator overloading is most useful when there is a logical mapping of a built-in operator to an operation on the user-defined type. It makes the program more natural and intuitive.
  - Operators should be used only for operations that are likely to be unambiguous to user. An operator had better not be overloaded if it plausibly has more than one interpretation.

11.2 Friends and friendship

- *Friend*s of a class have the same access to the class members as a member function. There are three basic varieties of friends: (1) friend non-member functions, (2) friend classes, and (3) friend methods, a.k.a. friend member functions.

  > Etymology: "friends" are intimate.

- Friend non-member functions of a class (NOT a member function)
  - syntax: prefix `friend` to the function prototype (NOT the outside definition) inside the class declaration. The prototype can be at ANY place of the class declaration.

    > Its declaration resides in the class declaration; yet it is NOT a class member.

  - A friend function's body can access ALL members of the target class. But this access privilege is NOT endowed to this friend function's argument list.
  - The declaration of the friend non-member function should be within the class. You should NOT redeclare it before or after.
  - An example

```
class Stock{
private: double price; ...
public: void operator*(unsigned float f){shares = shares * f;}
  ...
friend void operator*(unsigned float f, Stock &st);
friend void showPrice(double pr) {std::cout << pr << std::endl;}
};
void operator*(unsigned float f, Stock &st) {return st * f;}
// with friends, expressions "myStock*2" and "2*myStock" are both ok.
Stock st; showPrice(st.price); // error. "price" is inaccessible
```

  - At first glance, the friendship mechanism seems violated the OOP principle of data hiding because it allows non-member functions to access private and protected data. However that is an overly narrow view since friends are also a kind of interface for a class. Only a class declaration can decide which functions are friends, so the power of access control enjoyed be the class is not compromised.

- Fiend classes of a class
  - Syntax: declare the name of the friend class inside the target class's declaration: `friend class friendClassName;`, done. This declaration can be at ANY place inside the target class's declaration.

> Friendship is NOT mutual unless explicitly specified. *Mutual friendship* is discussed later in this section: "other friendly relationships".

- A friend class's methods' bodies can access ALL members of the target class. But this access privilege is NOT endowed to this friend class's method's argument list.
- The order of declarations is NOT arbitrary. The compiler has to have already known the basic information of items BEFORE their being used.

> It is much like the compiler should know a function or a variable's information before the function or variable is used - "declaring before using".
>
> Q: What "basic information" should the compiler know beforehand?
>
> A: For variables, the compiler must know their names and types (indicating their storage categories and sizes); for functions, the compiler must know their return types, names, and signatures. In short, the compiler has to gather the information on an item sufficient to generate a decorated name (see 8.4).

  - **Approach #1:** put the entire declaration of the friend class AFTER the declaration of the target class.

```cpp
class Tv{
friend class Remote; // declare a friend class called "Remote"
public: Tv(); void setChannel(int n); void toggleOnOff();
private: int channel; bool turnedOn; std::string brand;
};
class Remote{
public: Remote();
  void setChannel(Tv &t, int n) {t.setChannel(n);}
  void toggle(Tv &t) {t.toggleOnOff();}
};
```

  - **Approach #2: Firstly,** use a *forward declaration*. It has the effect of telling the compiler "`Tv` is a class". **Secondly,** because `Remote` class's methods uses `Tv` class's members, the compiler have to know these members' information beforehand. Therefore, you should also place the definitions of `Remote` class's methods after `Tv`'s class declarations.

```cpp
class Tv;    // forward declaration, not the real declaration.
class Remote {...}; // only member declarations
class Tv {friend class Remote; ...};
//... "Remote" class's methods are defined here
```

- Friend class is a natural idiom in which to express some kind of relationships. In the example of TV sets and remote controls above, were it not for this friendship mechanism, you would either have to make the private parts of the `Tv` class public, or construct some awkward, larger class that encompasses both a TV and a remote. And the latter workaround would not reflect the natural relationship between a TV set and a remote control.

- Friend methods, a.k.a. friend member functions, of a class
  - You can make a particular method of a class, instead of the whole class, a friend of the target class. This practice makes friendships more specific and thus less prone to error.

- The friend method's body (not its argument list) can access all the members of the target class.
- Syntax: declare the name of the friend method inside the target class's declaration: `friend returnType methodClass::methodName(argumentList);`, done. This declaration can be at ANY place inside the target class's declaration.

  > Note that the the name of the friend method should be qualified with the name of its own class, because this friend declaration statement is in the territory, namely the class scope, of the target class.

- As with friend classes, the order of declarations is NOT arbitrary. The compiler has to have already known the basic information of items BEFORE their being used.
  - **Erroneous approach:** this approach is NOT valid because when the compiler processes the two friend method declarations (line 2, 3), it has no way of knowing what `Remote` is (maybe it is a namespace) and whether `setChannel` and `toggle` are indeed in the scope of `Remote`.

    ```
     1  class Tv{
     2  friend void Remote::setChannel(Tv &t, int n);
     3  friend void Remote::toggle(Tv &t);
     4  public: Tv(); void setChannel(int n); void toggleOnOff();
     5  private: int channel; bool turnedOn; std::string brand;
     6  };
     7  class Remote{
     8  public: Remote();
     9    void setChannel(Tv &t, int n); void toggle(Tv &t);
    10  };
    11  inline void Remote::setChannel(Tv &t, int n) {t.setChannel(n);}
    12  inline void Remote::toggle(Tv &t) {t.toggleOnOff();}
    ```

  - **Correct Approach: Firstly,** use a *forward declaration* - the first line `class Tv;` tells the compiler " `Tv` is a class". **Secondly,** because in `Tv` class's declarations, the friend method declarations (line 4, 5) mentions `Reomote` class AND its members, you should put the declaration of `Remote` class before `Tv`'s. **Thirdly,** because `Remote` class's methods uses `Tv` class's members, the compiler have to know these members' information beforehand. Therefore, you should also place the definitions of `Remote` class's methods after `Tv`'s class declarations.

    ```
    1  class Tv;    // forward declaration, not the real declaration.
    2  class Remote {...}; // only declarations
    3  class Tv { ... // "Tv" members
    4  friend void Remote::setChannel(Tv &t, int n);
    5  friend void Remote::toggle(Tv &t);
    6  };
    7  //... "Remote" methods are defined here
    ```

    > The statement `friend class Remote;` itself suffices to tell the compiler "`Remote` is a class", but the statement `friend returnType Remote::some_func(...);` does NOT (because `Remote` might be a namespace).

- Other friendly relationships

- *Mutual friendship*. You can make classes friends to each other. One thing to keep in mind is that `A`'s methods that uses `B`'s members can be prototyped before `B` class declaration, but must be defined after (declaring-before-using rule).

  ```
  class A {friend class B; ...}; // only declarations
  class B {friend class A; ...};
  //...A's methods are defined here
  ```

  Mutual friendship is of usefulness when designing classes analogous to inter-communicating miscellaneous items, such as rover-satellite-earth systems, and Internet of Things.

  > Otherwise you may have to use awkward, counter-intuitive class inheritance to model a inter-communicating miscellaneous item network.

- *Shared friendship*. A function can be friends with more than one classes, by declaring the function a friend in each class. Similarly, you should pay attention to the "declaring before using" rule mentioned earlier in this section.
  - The shared friend can be a non-member function

    ```
    class ControlBase;
    class Satellite{ ...
    friend void clockSync(ControlBase &cb, Satellite &s);
    friend void clockSync(Satellite &s, ControlBase &cb);
    };
    class ControlBase{ ...
    friend void clockSync(ControlBase &cb, Satellite &s);
    friend void clockSync(Satellite &s, ControlBase &cb);
    };
    void clockSync(ControlBase &cb, Satellite &s) {...}
    void clockSync(Satellite &s, ControlBase &cb) {...}
    ```

  - The shared friend can be a member function of a third class.

    ```
    class Satellite; class ControlBase;
    class TimeService { // the time service center
    public: void clockSync(ControlBase &cb, Satellite &s) {...}
            void clockSync(Satellite &s, ControlBase &cb) {...}
    };
    class Satellite {
    friend void TimeService::clockSync(ControlBase &cb, Satellite &s);
    friend void TimeService::clockSync(Satellite &s, ControlBase &cb);
    };
    class ControlBase {
    friend void TimeService::clockSync(ControlBase &cb, Satellite &s);
    friend void TimeService::clockSync(Satellite &s, ControlBase &cb);
    };
    ```

  - The shared friend can be a third class.

```cpp
class Satellite; class ControlBase;
class TimeService {
public: void clockSync(ControlBase &cb, Satellite &s);
        void clockSync(Satellite &s, ControlBase &cb);
};
typedef TimeService TS; // you can use "typedef" with classes, too
class Satellite { friend class Channel; ... };
class ControlBase { friend class Channel; ... };
void TS::clockSync(ControlBase &cb, Satellite &s) {...}
void TS::clockSync(Satellite &s, ControlBase &cb) {...}
```

- Caution:
  - Friendship is not mutual unless explicitly specified.
  - Friendship cannot be inherited (just like man-kind, ironically).
  - Friendship cannot be transited. That is, if A is a friend of B and B is a friend of C, A is not guaranteed to be a friend of C (just like man-kind, ironically).

11.3 Conversion between built-in types and classes

- In C++, any constructor that takes a single argument can implicitly convert a value to the class type, just like a built-in type implicit conversion. This process is termed *implicit conversion*. *Explicit conversion* is also supported.

  ```cpp
  class Distance{
  private: double meters; double feet;
  public:
    Distance(double m) {meters = m; feet = 3.281 * meters;}
    Distance(double m, double f) {meters = m; feet = f;}
  };
  ```

  ```cpp
  Distance x = Distance(9.6); Distance y(9.6, 31.5); // ok
  Distance s = 9.6;   // 1-argument constructor allows implicit conversion
  s = 10.2;           // 1-argument constructor allows implicit conversion
  s = (Distance)11.2; // 1-argument constructor allows explicit conversion
  ```

  - Implicit conversions provided by the one-argument constructor is used for the following situations:
    - When you initialize or assign to a class type variable (as showed above);
    - When you pass a value to a function that expects a class type variable;

      ```cpp
      void display(Distance &dis) {...}; // use: display(10.2);
      ```

    - When you return a value in a function that is declared to return a class type variable.

      ```cpp
      Distance defualtSet() {...; return 10.2;}
      ```

    - When any of the preceding situations use a built-in type value that can unambiguously be converted to the constructor's argument type.

        - However, if you have multiple one-argument constructors that take in different types, the conversion is considered "ambiguous" and thus not allowed.

- Sometime the implicit conversion, as demonstrated above, is not always desirable. You can use keyword `explicit` to turn off implicit conversion, but STILL allows explicit conversion: `explicit Distance(double m) {...}`. *It is recommended.*

```
Distance s = (Distance)11.8; // explicit conversion
Distance z = 11.8;  // error. Because implicit conversion is turned off
```

- You can convert a value to the class type by using the class's one-argument constructor. Can you converse a class type to a built-in type? The answer is yes.
    - *Conversion function* is a kind of class method used to convert an object of that class to another type. Its prototype is `operator typeName();`
    - The `typeName` must be a built-in type name.

    > Built-in type names can be regarded as operators when used in type conversion (see 3.9). Prefixing `operator` suggests a conversion function is an instance of operator overloading (see 11.1), a special one (because it can be called implicitly, as discussed below).

    - As showed in its prototype, a conversion function has NO return type (NOT `void`), takes NO arguments, and has keyword `operator` prefixed to its declaration AND definition.

    ```
    class Distance {
    private: double meters; double feet;
    public:
        Distance(double m) { meters = m; feet = 3.281 * meters; }
        Distance(double m, double f) { meters = m; feet = f; }
        operator double(); // a conversion function
    };
    Distance::operator double() { return meters; }
    ```

    - You can use both explicit or implicit conversion syntaxes to invoke the conversion function.

    ```
    Distance s(9.6, 31.5);
    double a = (double)s;       // explicit
    double b = double(s);       // explicit
    double c = s;  // implicit call is also valid, let the compiler think
    ```

    - If you prefix keyword `explicit` to the conversion function's declaration (NOT definition): `explicit operator double();`, the implicit call becomes invalid. *It is recommended.*

☐