Lecture 10: Function and Operator Overload
Function overload

- C++ allows multiple functions of same scope (global, same namespace, static in same source file,...) having the same name, but different parameter lists (number of parameters or their types)
  1. int compare(int n1, int n2);
  2. int compare(float x1, float x2);
  3. bool compare(float x1, float x2); // error
  4. int compare(string& s1, string& s2);
  5. int compare(const string& s1, const string& s2);

- To determine the called function, the compiler prioritizes functions with same parameter types first, then finds the function with parameter types that given arguments can be casted to

```cpp
string ss1("xyz"), ss2("mpnq");
const string cs("aaa");
compare(1.3, 2.5); // error
compare("abcd", "12345"); // function 5
compare(ss1, ss2); // function 4
compare(ss1, cs); // function 5
```
Class method overload

- Similarly, methods in a class can be overloaded
  
  ```cpp
class C {
public:
  int compare(int x, int y);
  int compare(int x, int y) const;
  int compare(float x, float y);
};
```

- Overloading a method in child class will hide the method of same name in parent class
  
  ```cpp
class D: public C {
public:
  int compare(string s1, string s2);
};
```

D d;
d.compare("1234", "abcd");  // OK
d.compare(10, 20);          // error
d.C::compare(10, 20);       // OK
Default values of function/method parameters

- Parameters can have default values (which are values taken if none is given when being called)

- Parameters with default values must be last ones in parameter list
  
  ```
  void out(double x, int width = 7, int prec = 3) {...}
  out(1.2345, 10, 5);
  out(1.2345, 10);  // → out(1.2345, 10, 3);
  out(1.2345);     // → out(1.2345, 7, 3);
  ```

- Default values need only to be given in prototype
  
  ```
  double df(double x, int order = 1);
  // ...
  double df(double x, int order) {...}
  ```

- Possible to use expressions for default values, if they donot include other parameters of the same function
  
  ```
  UserProfile usr;
  double out(double x, int prec = getPrecOption(usr));
  double next(double x, double dx = diff(x));  // error
  ```
Default values of function/method parameters (cont.)

- Avoid confusion with overloaded functions
  - void input(double& x);
    void input(double& x, const char* prompt = "Enter x: ");
    input(y);  // error

- Default values of method parameters: similar to those of functions
  - class Vehicle {
      void out(int prec = 3);
  }

- Default values of constructor parameters
  - class Vehicle {
      public:
        Vehicle();  // default constructor
        Vehicle(Color c = Color::black, int wheels = 4);
    }
    Vehicle v1(Color::red);
    Vehicle v2(Color::white, 8);
    Vehicle v3;  // error

- Functions/methods with arbitrary number of parameters: self-study
Operator overload
General

- Operators in C++ can be redefined for new types
  - Ex: once Vector class is defined, we can define +, -, * operators to make following operations possible:
    
    ```
    Vector v1, v2, v3;
    v3 = -v1 + v2*2;    // expression using 4 operators
    ```
  - However, operators for built-in types cannot be redefined:
    ```
    int x = 3 + 2*5;
    double y = 2.54/1.23 + 3.11;
    ```

- To define operators, write a so-called operator function with corresponding parameters and return type
  - Operator functions can be global or class methods
  - Default values cannot be applied to operator-function parameters
  - If defined as a class method, the first operand is the object, and need not to be declared
General (cont.)

- Almost operators in C++ can be overloaded

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>-</td>
<td>*</td>
<td>/</td>
<td>%</td>
<td>^</td>
<td>&amp;</td>
<td></td>
<td></td>
<td>!</td>
</tr>
<tr>
<td>=</td>
<td>&lt;</td>
<td>&gt;</td>
<td>+=</td>
<td>-=</td>
<td>*=</td>
<td>/=</td>
<td>%=</td>
<td>^=</td>
<td>&amp;=</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>&lt;&lt;</td>
<td>&gt;&gt;</td>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td>==</td>
<td>!=</td>
<td>&lt;=</td>
<td>&gt;=</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>++</td>
<td>--</td>
<td>,</td>
<td>-&gt;*</td>
<td>-&gt;</td>
<td>()</td>
<td>[]</td>
</tr>
<tr>
<td>new[]</td>
<td>delete[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Some operators like + − * & have different meanings when being used as unary or binary, but both can be overloaded
- All above operators except =, when being defined in a class, are inherited by child classes
- A small number of operators cannot be overloaded:
  - . * :: ?: sizeof
- Operators can be overloaded, but their priority in expressions are fixed and cannot be changed
Overloading unary operators

- Defined by using global operator functions with one argument, or class methods without argument

Syntax:
- `<return type> operator <operator>(<type> <params>) {...}`
  
  or:

  ```
  class <class name> {
    <return type> operator <operator>() [const] {...}
  }
  ```

Example:
- `Vector operator -(const Vector& v)`
  ```
  { return Vector(-v.x, -v.y, -v.z); }
  ```

  or:

  ```
  class Vector {
    public:
      Vector operator -() const
      // argument is actually *this
      { return Vector(-x, -y, -z); }
  }
  ```
Overloading unary operators (cont.)

- Operator functions defined as global are usually declared with “friend” so that they can access to private members of the object
  ```cpp
  class Vector {
  public:
    friend Vector operator -(const Vector& v);
  };
  ```

  ```cpp
  Vector operator -(const Vector& v)
  { return Vector(-v.x, -v.y, -v.z); } 
  ```

- Example of using overloaded unary operators in expression:
  ```cpp
  Vector v1(1.2, 2.3), v2;
  v2 = -v1;
  ```

- Possible to explicitly call operator functions:
  ```cpp
  v2 = operator -(v1); // global operator function
  ```
  ```cpp
  or:
  v2 = v1.operator -(); // operator function as class method
  ```
Increment ++ and decrement -- operators

- These two operators can be put in front of (prefix) or after (postfix) the operand. To distinguish, prefix operator is defined just like others, but postfix one has a pseudo-argument of type int (even not used).

- Example:
  ```cpp
  class LimitedNum {
  private:
    int n, lim;
  public:
    LimitedNum& operator ++() { // prefix operator
      if (++n > lim) n = lim;
      return *this; }
    LimitedNum& operator ++(int) { // postfix operator
      return ++(*this); }
  }
  ```

- Explicit calls:
  ```cpp
  n.operator ++(); // calls prefix operator
  n.operator ++(0); // calls postfix operator
  ```

- Attn: similar for global operator functions
Type cast operators

- Similar to other unary operators, but can only be defined as class methods, and return types are implicit (no need to specify)
  - class Fraction {
    private:
      int a, b;
    public:
      operator double() { return (double)a/(double)b; } // Type casting to double
      operator string() { ... } // Type casting to string
      operator const char*() { ... } // Type casting to const char*
    ...
  }

- Usage:
  - Fraction f(4, 5);
    double d = (double)f + 1.2;
    string s(f);
    strcpy(cstr, f);

- Attn: make distinction between casting operators (class → other classes) and casting constructors (other classes → class)
Overloading binary operators

- Defined by using global operator functions with two arguments, or class methods without one argument

Example:

- Vector operator -(const Vector& v1, const Vector& v2)
  
  ```
  { return Vector(v1.x-v2.x, v1.y-v2.y, v1.z-v2.z); }
  ```

or:

- class Vector {
  public:
    Vector operator -(const Vector& v) const
      // first argument is *this
      { return Vector(x-v.x, y-v.y, z-v.z); }
  }

Example of using overloaded binary operator:

- v3 = v2-v1;

Similarly to unary operators:

- Usually declare global operator functions with “friend” to be able to access private class members
- It is possible to explicitly call operator functions
Comparison operators

- Examples:
  - class Vector {
    public:
    
    bool operator == (const Vector& v) const // class method
    {
        return x == v.x && y == v.y;
    }
    
    friend bool operator != (const Vector&, const Vector&);
  };

  // global function:
  
  bool operator != (const Vector& v1, const Vector& v2)
  {
      return !(v1 == v2); // reuse == operator
  }

- Other comparison operators can be defined in the same manner: > < >= <=
Assignment operators

- Can only be defined as class methods
  
  ```
  class Complex {
    public:
      Complex& operator =(const Complex& c);
      Complex& operator =(double x);
      Complex& operator +=(const Complex& c);
      Complex& operator -==(const Complex& c);
      Complex& operator *=(double x);
  }
  ```

- Other assignment operators are defined in the same manner:

  ```
  =  +=  -=  *=  /=  ^=  &=  |=  <<=  >>=
  ```
= operator

- Differences compared to other operators:
  - Also known as copy operator
  - If not defined, the compiler will generate a default one, which copies the member variables from the argument of same type
  - Is not inherited by child classes (actually, it is hidden by default copy operators of child classes)
- Make distinction with copy constructor
  - Vector v2(v1), v3 = v2; // both use copy constructor
  - v3 = v2; // copy operator
- Make distinction with casting constructors
  - string s1("12"), s2 = "ab"; // casting constructors
  - s2 = "xyz"; // copy operator
Summary on type casting manners

A a;
B b = a;

A::operator B(&)() [const]
B::B([const] A&o)
C::C([const] A&o)
A::operator C(&)() [const]
B::B([const] C&o)

A B
B::B([const] A&o)
A::operator C(&)() [const]
C::C([const] A&o)
B::B([const] C&o)
Summary on type casting manners

```cpp
void fff([const] B[&] b);

A a;
fff(a);
```

![Diagram](diagram.png)
Summary on type casting manners

A a;
B b;
b = a;

A::operator B[&]() [const]
B::B([const] A[&])
B& B::operator =([const] B[&])
C
A::operator B[&]() [const]
B& B::operator =([const] C[&])
A::operator C[&]() [const]
B::C([const] A[&])
C::C([const] A[&])
B& B::operator =([const] C[&])
new, new[], delete, and delete[] operators

- Used to allocate and deallocate memory blocks from heap
- Attn: constructor and destructor calls are made automatically, cannot be altered

```cpp
class Obj {
public:
    void* operator new(size_t sz) {
        return malloc(sz);
    }
    void* operator new[](size_t sz) {
        return malloc(sz);
    }
    void operator delete(void* p) {
        free(p);
    }
    void operator delete[](void* p) {
        free(p);
    }
};
```
Other special operators: self-study

- Function call: $p(x, y)$
- Index (subscript): $\text{arr}[i]$
- Comma: $a, b$
- Indirection (dereference): $*\text{ptr}$
- Structure dereference: $\text{pnt}->\text{mem}$
- Pointer to a member: $\text{obj}->*\text{mem}$
- Placement new: $\text{new } (p)[n]$
cout, cin and input/output operators

- cout, cin are instances of ostream and istream classes, in which << and >> are overloaded for input and output

- Example: (*)
  
  ```
  ostream& operator <<(int x) {...}
  ostream& operator <<(float x) {...}
  ostream& operator <<(double x) {...}
  ostream& operator <<(char x) {...}
  ostream& operator <<(const char* s) {...}
  ...
  istream& operator >>((int& x) {...}
  istream& operator >>((float& x) {...}
  istream& operator >>((double& x) {...}
  istream& operator >>((char& x) {...}
  istream& operator >>((char* s) {...}
  ...
  ```

* These examples are only for illustration. In reality, ostream and istream classes are defined and implemented with a slight difference. See more in the chapter about STL.
Overloading $\langle\langle$ and $\rangle\rangle$ operators for input/output

- Overload $\langle\langle$ and $\rangle\rangle$ operators for any classes to make input/output with cin, cout possible with their instances
  - class Vector {
    // declare "friend" for operators
  };
  
  ostream& operator $\langle\langle$(ostream& s, const Vector& v) {
    s $\langle\langle' ( $\langle\langle v.x $\langle\langle ', ' $\langle\langle v.y $\langle\langle ', ' $\langle\langle v.z $\langle\langle ');
    return s;
  }
  
  istream& operator $\rangle\rangle$(istream& s, Vector& v) {
    s $\rangle\rangle v.x $\rangle\rangle v.y $\rangle\rangle v.z;
    return s;
  }

- Using overloaded operators:
  - Vector v1, v2;
    cout $\langle\langle "$v1 = "$ $\langle\langle v1;
    cin $\rangle\rangle v2;
Problems

1. Define all operators needed for Vector class: add, subtract, multiply with number, dot product, cross product
2. Define operators needed for Complex class
3. Define operators needed for String class: + (with character or string), type casting, input/output, [ ] (element at)
4. Write BigInt class to manipulate arbitrary big integers and define needed operators: +, -, *, /, ++, --, casting to string/long long
5. Write Array class to manipulate dynamic arrays of integers: << (add element, merge two arrays), [ ], type casting
6. Write Iterator class to traverse linked lists by ++ (move to next element), ! (check if reached end of list), * (obtain object at current position) operators. Then define ~ (create Iterator instance) operator for LList. The goal is that, after above definitions, it is possible to traverse a linked list by:

```cpp
LList lst;
for (Iterator itr = ~lst; !itr; itr++) {
    int& data = *itr;
    // ...
}
```