COMP 2012H Honors Object-Oriented Programming and Data Structures

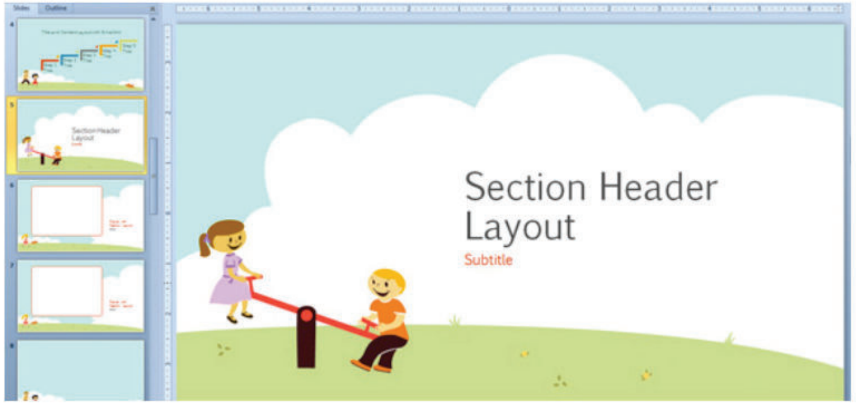
Topic 13: Generic Programming

Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Part I

Function and Class Template



How Many my_max() Functions Do You Need?

```
int my_max(const int& a, const int& b)
    { return (a > b) ? a : b; }

char my_max(const char& a, const char& b)
    { return (a > b) ? a : b; }

double my_max(const double& a, const double& b)
    { return (a > b) ? a : b; }

#include <string>
string my_max(const string& a, const string& b)
    { return (a > b) ? a : b; }

#include "teacher.h"
Teacher my_max(const Teacher& a, const Teacher& b)
    { return (a > b) ? a : b; }
```


How Many Stack Classes Do You Need?

```
class int_stack {
  private: int data[100]; int top_index;
  public: int_stack();
    int top() const; void push(int); void pop();
    bool empty() const; bool full() const; int size() const;
};

class char_stack {
  private: char data[100]; int top_index;
  public: char_stack();
    char top() const; void push(char); void pop();
    bool empty() const; bool full() const; int size() const;
};

#include "student.h"
class student_stack {
  private: Student data[100]; int top_index;
  public: student_stack();
    Student top() const; void push(Student); void pop();
    bool empty() const; bool full() const; int size() const;
};
```

Generic Programming using Templates

- A lot of times, we find functions and data structures that look alike: they differ only in the types of objects they manipulate.
- Since C++ allows function overloading, one may define many **my_max()** functions, one for each type of values/objects **T**, but they all have the following general form:

  T my_max(const T& a, const T& b) { . . . }
- For **stack**s of different types of objects, one has to make up different class names for them (int_stack, char_stack, etc.).
- Again, we don't like the solution of creating the various **my_max()** or **stack**s by "copy-and-paste-and-modify".
- The solution is generic programming using function templates and class templates.
- They are similar to function definitions and class definitions but the types of objects they manipulate are parameterized with type variables.
- Generic programming allows programmers to write just one version of code that works for different types of objects.

Function Template of my_max( )

- It starts with the keyword template.

```
template <typename T>    /* File: max-template.cpp */
T my_max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}
```

- The typename keyword may be replaced by class.

```
template <class T>       /* File: max-template2.cpp */
T my_max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}
```

- This is just a function template definition; it itself is not an actual function and no codes will be generated on its own.

Use of the my_max( ) Function Template

- You may make use of the template to call **my_max()** for any types, as long as the function code makes sense for the types.
- In the case of **my_max()**, it is required that the types can be compared by the operator " >".

```
#include <iostream>      /* File: max-calls.cpp */
using namespace std;
template <typename T>
T my_max(const T& a, const T& b) { return (a > b) ? a : b; }

int main()
{
    int x = 4, y = 8;
    cout << my_max(x, y) << " is a bigger number!" << endl;

    string a("cheetah"), b("gorilla");
    cout << my_max(a, b) << " is stronger!" << endl;
    return 0;
}
```


Function Template Instantiation

- Based on the function template definition, the compiler will create the codes of the functions that are actually used (called) in your program.
- This is called template instantiation. The parameter **T** in the template definition is called the formal parameter or formal argument of the template.
- For the program "max-calls.cpp", the compiler will instantiate 2 **my_max()** functions by substituting **T** with the actual arguments **int** and **string** respectively into the my_max function template.

```
template <typename T>
T my_max(const T& a, const T& b) { return (a > b) ? a : b; }
```

Template: Formal Argument Matching

```
#include <iostream>     /* File: max-match-arg.cpp */
using namespace std;
template <typename T>
T my_max(const T& a, const T& b) { return (a > b) ? a : b; }

int main()
{
    cout << my_max(3, 5) << endl;     // T is int;
    cout << my_max(4.3, 5.6) << endl; // T is double
}
```

- When the compiler instantiates a template, it tries to determine the actual type of the template parameter by looking at the types of the actual arguments in a function call.
- If you call a template with different types, the compiler will generate separate instantiated function code for each type, and the size of the final executable increases accordingly.

Explicit Template Instantiation

- However, there is no automatic type conversion for template arguments.
- The following code gives a compile-time error:

```
cout << my_max(4, 5.5);
// Error: no matching function for call to 'my_max(int, double)'
```

- If what you really want is:

```
double my_max(const double& a, const double& b) { ... }
```

  you may do this by explicitly instantiating the function template by adding the actual type you want after the function name using the < > syntax:

```
cout << my_max<double>(4, 5.5);
```


Function Template w/ More Than One Formal Argument

```
#include <iostream>     /* File: fcn-template-2arg.cpp */
using namespace std;

template <typename T1, typename T2>
T1 my_max(const T1& a, const T2& b) { return (a > b) ? a : b; }

int main()
{
    cout << my_max(4, 5.5) << endl; // T1 is int, T2 is double
    cout << my_max(5.5, 4) << endl; // T1 is double, T2 is int
}
```

- A template may take more than one type arguments, each using a different typename.
- However, there is a subtle problem in this case: the return type of this **my_max** is the type of the first argument.
- So what will the above code print?

Function Template w/ More Than One Formal Argument ..

- The following template definition does not suffer from the problem but it doesn't return a value.

```
#include <iostream>     /* File: fcn-template-2arg-ok.cpp */
using namespace std;

template <typename T1, typename T2>
void print_max(const T1& a, const T2& b)
{
    if (a > b)
        cout << a << endl;
    else
        cout << b << endl;
}

int main() { print_max(4, 5.5); print_max(5.5, 4); }
```

Template Arguments: Too Many Combinations

```
/* File: many-combinations.cpp */
short s = 1; char c = 'A';
int i = 1023; double d = 3.1415;

print_max(s, s); print_max(s, c);
print_max(c, s); print_max(s, i);
// ... And all other combinations; 16 in total.
```

- With the above code, the compiler will instantiate a **print_max()** for each of the 16 different combinations of arguments.
- With the current compiler technology, this means that we get 16 (almost identical) fragments of code in the executable program. There is no sharing of code.
- So a simple program may have a surprisingly large binary size, if we are not careful.

Function Template: Common Errors

```
1  #include <iostream>      /* File: f-template-err.cpp */
2  using namespace std;
3  template <class T> T* create() { return new T; };
4  template <class T> void f() { T a; cout << a << endl; }
5  int main() { create(); f(); }
```

```
f-template-err.cpp:5:21: error: no matching function for call to 'create()'
    int main() { create(); f(); }
                        ^
f-template-err.cpp:3:23: note: template argument deduction/substitution failed
f-template-err.cpp:5:21: note: couldn't deduce template parameter 'T'
    int main() { create(); f(); }
                        ^
f-template-err.cpp:5:26: error: no matching function for call to 'f()'
    int main() { create(); f(); }
                             ^
f-template-err.cpp:4:25: note: template argument deduction/substitution failed
f-template-err.cpp:5:26: note: couldn't deduce template parameter 'T'
    int main() { create(); f(); }
```

The compiler can't deduce the actual object types from such calls.

Class Template for Nodes of a List

- The template mechanism works for classes as well. This is particularly useful for defining container classes — classes that contains objects of the same kind such as arrays, lists, and sets.

```
#ifndef LISTNODE_H      /* File: listnode.h */
#define LISTNODE_H

template <typename T>
class List_Node
{
  public:
    List_Node(const T& x) : data(x) { }

    List_Node* next {nullptr};
    List_Node* prev {nullptr};
    T data;
};

#endif
```


Class Template for a List

```
#ifndef LIST_H          /* File: list.h */
#define LIST_H

#include "listnode.h"
template <typename T> class List
{
  public:
    List() = default;
    void append(const T& item) {
        List_Node<T>* new_node = new List_Node<T>(item);
        if (!tail)
            head = tail = new_node;
        else
        { /* incomplete */ }
    }
    void print() const {
        for (const List_Node<T>* p = head; p; p = p->next)
            cout << p->data << endl;
    }
    // ... Other member functions
  private:
    List_Node<T>* head {nullptr};
    List_Node<T>* tail {nullptr};
};
#endif
```

Class Template: List Example

- Now we can use the parameterized class template **list** to create lists to store any types of elements that we want, without having to resort to "code re-use by copying".

```
#include <iostream>      /* File: list-example.cpp */
using namespace std;
#include "list.h"
#include "student.h"

int main()
{
    List<char> letters; letters.append('a');
    cout << "*** print char list *** \n"; letters.print();

    List<int> primes; primes.append(2);
    cout << "### print int list ###\n"; primes.print();

    List<Student> students;
    students.append(Student("James", CSE, 4.0));
    // Why don't we call students.print() ?
}
```


Nontype Parameters for Templates

- Template may also have nontype parameters, which are not type variables.

```
#ifndef NONTYPE_LIST_H  /* File: nontype-list.h */
#define NONTYPE_LIST_H
#include "listnode.h"
template <typename T, int max_num_items>
class List {
  public:
    bool append(const T& item) {
        if (num_items == max_num_items)
           { cerr << "List is full\n"; return false; }
        else
           { /* incomplete */ return true; }
    }
    // ... Other member functions
  private:
    int num_items {0};
    List_Node<T>* head {nullptr};
    List_Node<T>* tail {nullptr};
};
#endif
```


Difference Between Class and Function Templates

- For function templates, the compiler may deduce the template arguments from the function call.

```
int i = my_max(4, 5); // Rely on compilers to deduce my_max<int>
int j = my_max<int>(7, 2); // Explicit instantiation
```

- For class templates, you always have to specify the actual template arguments when creating the class objects; the compiler does not deduce the template arguments.

```
List primes;      // Error: how can compilers deduce the type?
primes.append(2); // Error: too late; compilers can't lookahead!
```

Which scene is slightly different from the other two?

Separate Compilation For Templates??

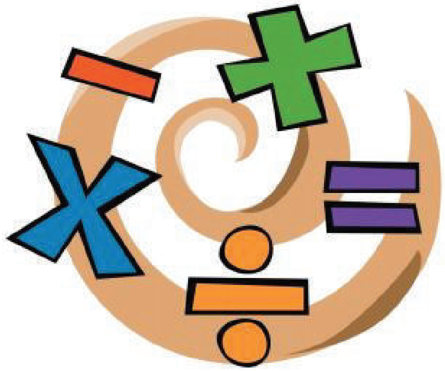
- For regular non-template functions, we usually put their declarations in a header file, and their definitions in the corresponding .cpp file.
- Should we do the same for templates?

```
/* File: max.h */
template <typename T> T my_max(const T& a, const T& b);

/* File: max.cpp */
template <typename T> T my_max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}
```

  But a function/class template is instantiated only when it is used, and its definition must be in the same file which calls it.
- No, we put the template function/class definitions in the header file as well and include the template header file in every files which use the template.

Part II

+*-/ Operator Overloading <&% >



From Math Notation to Language Operators

- To program the mathematical equation:

  ```
  c = 2(a - 3) + 5b
  ```

  one may have to write

  ```
  c = add(multiply(2, subtract(a,3)), multiply(5,b));
  ```

- Most programming languages have operators which allow us to mimic the mathematical notation by writing

  ```
  c = 2*(a - 3) + 5*b;
  ```

- However, many languages only have operators defined for the built-in types.
- C++ is an exception: it allows you to re-use most, but not all, of its operators and re-define them for new user-defined types.
- You may re-define "+", "-" etc. for types such as Vector, Matrix, Student, Word, etc. defined by you.

Add 2 Vectors by a Global Add( ) Function

```
using namespace std;     /* File: vector0.h */
class Vector
{
  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }
    double getx() const { return x; }
    double gety() const { return y; }
    void print() const { cout << "(" << x << ", " << y << ")\n"; }
  private:
    double x, y;
};
```

```
#include <iostream>      /* File: vector0-add.cpp */
#include "vector0.h"
Vector add(const Vector& a, const Vector& b)
    { return Vector(a.getx() + b.getx(), a.gety() + b.gety()); }
int main()
{
    Vector a(1, 3), b(-5, 7), c(22), d;
    d = add(add(a, b), c); d.print(); // d = a + b + c
}
```

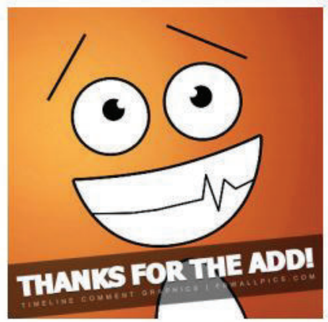

Global Non-member Operator+ Function

- Wouldn't it be nicer if we could write the last addition expression as: $d = a + b + c$ instead of

  ```
  d = add(add(a, b), c));
  ```

- C++ allows you to do that by simply replacing the name of the function **add** by **operator+**.
- Also notice that our global non-member **operator+** function will work for adding
  - a vector to a vector
  - a vector to a scalar
  - a scalar to a vector

Question: Why do they work?



Global Non-member Operator+ Function ..

```
#include <iostream>     /* File: vector0-op-add.cpp */
#include "vector0.h"

Vector operator+(const Vector& a, const Vector& b)
    { return Vector(a.getx() + b.getx(), a.gety() + b.gety()); }

int main()
{
    Vector a(1, 3), b(-5, 7), c(22), d;
    d = a + b + c; cout << "vector + vector: a + b + c = ";
    d.print();

    d = b + 1.0; cout << "vector + scalar: b + 1.0 = ";
    d.print();

    d = 8.2 + a; cout << "scalar + vector: 8.2 + a = ";
    d.print();
    return 0;
}
```


Operator Function Syntax

- **operator+** is a formal function name that can be used like any other function name.
- We could have called the **operator+** function in the formal way as

  `d = operator+(operator+(a, b), c);`

  But who would want to write code like that?
- Operator functions in C++ are just like ordinary functions, except that they also can be called with a nicer syntax similar to the usual mathematical notations.
- The operator + has a formal name, namely **operator+** (consisting of 2 keywords), and a "nickname," namely +.
- The formal name requires you to call it as

  `operator+(a, b)`

  while the simple nickname let you call it as

  `a + b`

Operator Syntax . . .

- The nickname can only be used when calling the function.
- The formal name can be used in any context, when declaring the function, defining it, calling it, or taking its address.
- There is nothing that you can do with operators that cannot be done with ordinary functions. In other words, operators are just syntactic sugar.
- Be careful when defining operators. There is nothing that inhibits you from coding **operator+** to do, e.g., subtraction.
- Similarly, nothing inhibits you from defining **operator+** and **operator+=** so that the following 2 expressions: `a = a + b` and `a += b`, have 2 different meanings.
- However, your code will become unreadable.

Don't shock the user!

C++ Operators

- Almost all operators in C++ can be overloaded except:
  `.` `::` `?:` `.*` (reason)
- The C++ parser is fixed. That means that you can only re-define existing operators, and you cannot define new operators (using new symbols).
- Nor can you change the following properties of an operator:

1. Arity: the number of arguments an operator takes.
   e.g., `!x` `x+y` `a%b` `s[j]`
   (So you are not allowed to re-define the + operator to take 3 arguments instead of 2.)
2. Associativity: e.g. `a+b+c` is always identical to `(a+b)+c`.
3. Precedence: which operator is done first?
   e.g., `a+b*c` is treated as `a+(b*c)`.

C++ Operators: Member or Non-member Functions

- All C++ operators already have predefined meaning for the built-in types. It is impossible to change their meaning.
- You can only overload operators for your own (user-defined) classes (such as **Vector** in the example above) with new meanings.
- Therefore, every operator function you define must implicitly have at least one argument of a user-defined class type.
- You may define a (new) operator function as a member function of a new class, or as a global non-member function.
- As a global function, **operator+** has 2 arguments. When it is called in an expression such as **a + b**, it is equivalent to writing **operator+(a, b)**.
- More about defining operator function as a member function of a class later.

Global Non-member Operator<< Function

```
void print() const { cout << "(" << x << ", " << y << ")\n"; }
```

- Until now, one prints out a **Vector** object by calling its **print** function.
- Let's write a non-member **operator**<< function to print **Vector** objects more naturally by using **cout** or **cerr**.
- The syntax should be similar to the one we use to print values of the basic types (such as **int**). E.g., `cout << x;`
- But **cout** and **cerr** are objects of the **ostream** class. So let's generalize the **operator**<< function to print **Vector**s to any **ostream** objects.
- **ostream** is the base class for all possible output streams.
- To allow the usual output syntax with **cout** on the left, the **ostream** object must be the first argument in the function.

Question: Why does it return **ostream&**?

Global Non-member Operator<< Function ..

```
#include <iostream>      /* File: vector0-op-add-os.cpp */
#include "vector0.h"
using namespace std;

ostream& operator<<(ostream& os, const Vector& a)
    { return (os << '('  << a.getx() << ", " << a.gety() << ')'); }

Vector operator+(const Vector& a, const Vector& b)
    { return Vector(a.getx() + b.getx(), a.gety() + b.gety()); }

int main()
{
    Vector a(1.1, 2.2);
    Vector b(3.3, 4.4);
    cout << "vector + vector: a + b = " << a + b << endl;
    cout << "vector + scalar: b + 1.0 = " << b + 1.0 << endl;
    cout << "scalar + vector: 8.2 + a = " << 8.2 + a << endl;
    return 0;
}
```


Global Non-member Operator<< Function ...

- The **operator**<< returns an ostream object because we like to cascade outputs in one statement such as:

```
Vector a(1, 0);
cout << " a = " << a << "\n";
```

- The second line is equivalent to:

```
operator<<(operator<<(operator<<(cout," a = "),a),"\n");
```

- This can only work if **operator**<< returns the ostream object itself.

Question: Could we define **operator**<< as a member function?

Operator+ Member Function

- Member operator functions are called using the same "dot syntax" by specifying an object of, for example, type **Vector**.
- If **a** is a **Vector** object, then the expression **a+b** is equivalent to **a.operator+(b)**.
- To call the **operator+** as a member function, the class object must be the left operand. (Here **a**.)
- Thus, when we define **operator+** as a member function of **Vector**, it has only one argument — the first argument is implicitly the object on which the member function is invoked.
- Recall the implicit this pointer in all member functions. Thus,

```
Vector operator+(const Vector& b) const;
```

of the class **Vector** will be compiled into the following global function:

```
Vector Vector::operator+(const Vector* this, const Vector& b);
```


Operator+ and Operator+= Member Functions

```
#include <iostream>      /* File: vector-op-add.h */
class Vector
{
  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }
    double getx() const { return x; }
    double gety() const { return y; }
    Vector operator+(const Vector& b) const;
    const Vector& operator+=(const Vector& b);
  private:
    double x, y;
};

Vector Vector::operator+(const Vector& b) const
{   // Return by value; any copy constructor?
    return Vector(x + b.x, y + b.y);
}
const Vector& Vector::operator+=(const Vector& b)
{
    x += b.x; y += b.y;
    return *this; // Return by const reference. Why?
}
```


Operator+ and Operator+= Member Functions ..

```
#include "vector-op-add.h" /* File: vector-op-add-test.cpp */
using namespace std;

ostream& operator<<(ostream& os, const Vector& a)
{
    return (os << '('  << a.getx() << ", " << a.gety() << ')');
}

int main()
{
    Vector a(1.1, 2.2);
    Vector b(3.3, 4.4);

    cout << "vector + vector: a + b = " << a + b << endl;
    cout << "vector + scalar: b + 1.0 = " << b + 1.0 << endl;
    cout << "scalar + vector: 8.2 + a = " << 8.2 + a << endl; //Error

    a += b;
    cout << "After += : a = " << a << "  b = " << b << endl;
    return 0;
}
```


Operator+ Member Function: Commutative?

- Whenever the compiler sees an expression of the form **a+b**, it converts the expression to the two possible representations

  operator+(a, b)
  a.operator+(b)

  and verifies whether one of them is defined.
- It is an error to define both.
- In math, we expect **operator+** to be commutative: $a + b$ is equivalent to $b + a$. Thus, we expect we may do (vector + scalar) and (scalar + vector) too.
- However, as a Vector member function, the left operand of **operator+** is always a Vector.
- The current version only works for (vector + vector) and (vector + scalar). Why?

Question: Why **operator+** and **operator+=** have different return types?

Operator+ (Vector, Scalar)

```
#include "vector-op-add.h" /* File: vector-op-add-ok.cpp */
using namespace std;

ostream& operator<<(ostream& os, const Vector& a)
    { return (os << '('  << a.getx() << " , " << a.gety() << ')'); }

int main()
{
    Vector a(1.1, 2.2);
    cout << "vector + scalar: a + 5 = " << a + 5 << endl;
}
```

- It works because the argument to the right of + which is a scalar can be converted to a Vector object.

Question: Where is the conversion constructor?

- Thus, the expression `(a + 5)` is converted to

  `a.operator+(Vector(5))`

Operator+ (Scalar, Vector)

- Let's do the other way: add a **Vector** object to a scalar.

```
1   #include "vector-op-add.h" /* File: vector-op-add-error.cpp */
2   using namespace std;
3
4   ostream& operator<<(ostream& os, const Vector& a)
5       { return (os << '('  << a.getx() << " , " << a.gety() << ')'); }
6
7   int main()
8   {
9       Vector a(1.1, 2.2);
10      cout << "scalar + vector: 5 + a = " << 5 + a << endl;
11  }
```

```
vector-op-add-error.cpp:10:46: error: no match for operator+
  (operand types are int and Vector)
     cout << "scalar + vector: 5 + a = " << 5 + a << endl;
                                              ~~^~~
```


Operator+ (Scalar, Vector): What's the Problem?

- Isn't the **operator+** commutative?
  Isn't the expression `(5 + a)` equivalent to `(a + 5)`?

  Yes, we expect they are. But `(5 + a)` will be converted to
  5.operator+(a)

  and **int** is not a class — there is no **operator+** member function for **int** nor can we re-define it.

- Wouldn't **5** be converted to a Vector object by Vector's conversion constructor and the result calls its **operator+** member function with argument **Vector a**?

  No, compilers will not try to do that. In theory, the scalar 5 can be possibly converted to objects of many user-defined classes if they have such conversion constructor. It will be a lot of work for a compiler to check all those possibilities, make the conversion, and then check if they can be added to a **Vector** object.

Non-member Operator+ (Scalar, Vector)

- One solution is to write a global non-member **operator+** whose first argument is a scalar, and the function actually calls the **operator+** member function of its 2nd Vector argument.

```
Vector operator+(double a, const Vector& b) { return b + a; }
```

- A better solution is our previous global non-member **operator+** function which takes 2 Vector arguments (if Vector class provides the public getx() and gety() functions to access x and y).

```
Vector operator+(const Vector& a, const Vector& b)
    { return Vector(a.getx()+b.getx(), a.gety()+b.gety()); }
```

Overload Operator= for Member Assignment

```
#include <iostream>     /* File: vector-op=.h */

class Vector
{
  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }
    const Vector& operator=(const Vector& b);
  private:
    double x, y;
};

const Vector& Vector::operator=(const Vector& b)
{
    if (this != &b) // Avoid self-assignment to save time
    {
        x = b.x;
        y = b.y;
    }
    return *this; // Why return const Vector& ?
};
```


Member Operator= with Owned Data Members

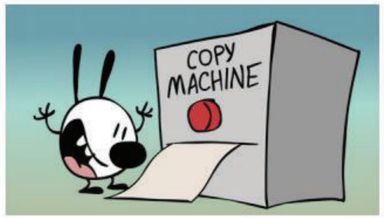
```
class Word               /* File: word.h */
{
  private:
    int freq; char* str;
    void setstr(const char* s)
        { str = new char[strlen(s)+1]; strcpy(str,s); }
  public:
    // The following str{nullptr} is necessary. Why?
    Word(const Word& w): str{nullptr} { cout << "Copy: "; *this = w; }
    Word(const char* s, int k = 1) : freq(k)
        { cout << "Conversion: from \"" << s << "\"\n"; setstr(s); }

    const Word& operator=(const Word& w) {
        if (this != &w)
        {
            cout << "op= with " << w.str << endl;
            freq = w.freq; delete [] str; setstr(w.str);
        }
        return *this;
    }
};
```


Member Operator= with Owned Data Members ..

```
#include <iostream>     /* File: word-test.cpp */
using namespace std;
#include "word.h"

int main()
{
    Word ship("Titanic");       // Which constructor?
    Word movie(ship);           // Which constructor?
    Word song("My heart will go on"); // Which constructor?

    song = song;                // Call assignment operator
    song = movie;               // Call assignment operator
}
```

Member Operator= with Owned Data Members ...

- If a class contains pointer data members and dynamic memory allocation is required, the default memberwise assignment — shallow copy — is not adequate.
- The copy constructor and **operator=** should be implemented so that each object has its own copy of the "owned" data.
- Since the copy constructor and **operator=** usually do the same thing, they may be defined by making use of the other.
- Here, the copy constructor is defined by calling **operator=**.



Member Operator[ ] To Access Vector Component

```
1  #include <iostream>     /* File: vector-op-index.h */
2  using namespace std;
3  class Vector {
4    public:
5      Vector(double a = 0, double b = 0) : x(a), y(b) { }
6      double operator[](int) const; // Read-only; c.f. getx() and gety()
7      double& operator[](int);      // Allow read and write
8    private:
9      double x, y;
10 };
11 double Vector::operator[](int j) const {
12     switch (j) {
13         case 0: return x;
14         case 1: return y;
15         default: cerr << "op[] const: invalid dimension!\n"; } }
16
17 double& Vector::operator[](int j) {
18     switch (j) {
19         case 0: return x;
20         case 1: return y;
21         default: cerr << "op[]: invalid dimension!\n"; } }
```


Member Operator[ ] To Access Vector Component ..

```
1  #include "vector-op-index.h" /* File: vector-op-index-test.cpp */
2
3  // Replace getx(), gety() by op[]
4  ostream& operator<<(ostream& os, const Vector& a) // Which op[]?
5  {
6      return (os << '('  << a[0] << " , " << a[1] << ')');
7  }
8
9  int main()
10 {
11     Vector a(1.2, 3.4);
12     cout << "Before assignment: " << a << endl;
13
14     a[0] = 5.6; a[1] = 7.8;   // Which op[]?
15     cout << "After assignment:  " << a << endl;
16
17     a[2] = 9;                 // Which op[]? Error!
18     return 0;
19 }
```


Why 2 Versions of Member Operator[ ]?

- Try to compile “vector-op-index-test.cpp” with only having the 2nd version of **operator[ ]**.

```
vector-op-index-test.cpp:6:28: error: no viable overloaded
      operator[] for type 'const Vector'
    return (os << '('  << a[0] << " , " << a[1] << ')');
                          ~^~
./vector-op-index.h:17:17: note: candidate function not viable:
      'this' argument has type 'const Vector', but method
      is not marked const
double& Vector::operator[](int j) {
```

- Try to compile “vector-op[ ]-test.cpp” with only having the 1st version of **operator[ ]**.

```
vector-op-index-test.cpp:14:10:
  error: expression is not assignable
    a[0] = 5.6; a[1] = 7.8; // Which op[]?
    ~~~~ ^
```


Member Operator++

```
class Vector {          /* File: vector-op-incr.h */
  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }
    double operator[](int) const; // Read-only; c.f. getx() and gety()
    double& operator[](int);      // Allow read and write
    Vector& operator++();         // Pre-increment returns an l-value
    Vector operator++(int);       // Post-increment returns a r-value
  private:
    double x, y;
};

Vector& Vector::operator++() { ++x; ++y; return *this; }

// The dummy must be an int argument. Why is it needed?
Vector Vector::operator++(int)
{
    Vector temp(x,y);
    x++; y++; return temp;
}

/* Plus the operator[] function definitions not shown here */
```

Member Operator++ ..

```
#include <iostream>     /* File: vector-op-incr-test.cpp */
#include "vector-op-incr.h"
using namespace std;

ostream& operator<<(ostream& os, const Vector& a)
    { return (os << '('  << a[0] << " , " << a[1] << ')'); }

int main()
{
    Vector a(1.1, 2.2);
    Vector b(3.3, 4.4);
    Vector c;

    c = ++a;
    cout << "a = " << a << "\nc = " << c << endl;

    c = b++;
    cout << "b = " << b << "\nc = " << c << endl;
    return 0;
}
```


Summary: Member or Non-member Operator Functions

- The operators: = (assignment), [ ] (indexing), ( ) (call) are required by C++ to be defined as class member functions.
- A member operator function has an implicit first argument of the class. Thus, if the left operand of an operator must be an object of the class, it can be a member function.
- If the left operand of an operator must be an object of other classes, it must be a non-member function. e.g., **operator**<<.
- For commutative operators like + and *, it is usually preferred to be defined as non-member functions to allow automatic conversion of types using the conversion constructors.

```
string x("dot"), y("com"), z;
z = x + y;
z = x + "com";
z = "dog" + y;
```


Part III

Friend Functions or Classes

Operator<< as a Member Function

- Let's try to implement **operator**<< as a member function.

```
#include <iostream>     /* File: vector-os-nonfriend.h */

class Vector
{
  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }
    double getx() const { return x; }
    double gety() const { return y; }
    ostream& operator<<(ostream& os);

  private:
    double x, y;
};

ostream& Vector::operator<<(ostream& os)
{
    return (os << '('  << x << " , " << y << ')');
}
```

Operator<< as a Member Function

```
#include <iostream>     /* File: vector-os-nonfriend.cpp */
using namespace std;
#include "vector-os-nonfriend.h"

Vector operator+(const Vector& a, const Vector& b)
    { return Vector(a.getx() + b.getx(), a.gety() + b.gety()); }

int main()
{
    Vector a(1.1, 2.2);
    Vector b(3.3, 4.4);
    Vector d = a + b;

    // Do you notice the strange output syntax?
    d << (cout << "vector + vector: a + b = ") << endl;
    (b + 1.0) << (cout << "vector + scalar: b + 1.0 = ") << endl;
    (8.2 + a) << (cout << "scalar + vector: 8.2 + a = ") << endl;
}
```


Issues of Operator<< as a Member Function

- **operator**<< is a binary operator. As a member function, the Vector object must be on the left of << and **cout** on the right.
- To print a Vector **x**, now you have to write: `x << cout;`
- Furthermore, to cascade outputs, say, to print Vectors **x**, **y** and then **z**, now you will have to write:

  `z << (y << (x << cout));`

  instead of the usual output syntax: `cout << x << y << z;`
- For such kinds of operators, it is better to implement them as global non-member functions.
- Two issues:
  1. Since global non-member functions can't access private data members, don't forget to provide the latter with public assessor member functions.
  2. However, non-member operators are less efficient due to the additional calls to assessor functions.
- A solution: Making friends!

Friend Member Operator<<

```
#include <iostream>     /* File: vector-with-friends.h */
using namespace std;

class Vector
{
    friend ostream& operator<<(ostream& os, const Vector& a);
    friend Vector operator+(const Vector& a, const Vector& b);

  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }

  private:
    double x, y;
};

ostream& operator<<(ostream& os, const Vector& a)
    { return (os << '('  << a.x << " , " << a.y << ')'); }

Vector operator+(const Vector& a, const Vector& b)
    { return Vector(a.x + b.x, a.y + b.y); }
```


Friend Member Operator<<

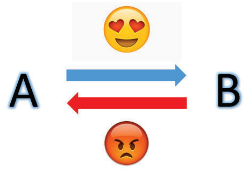
```
#include "vector-with-friends.h" /* File: vector-with-friends.cpp */

int main()
{
    Vector a(1.1, 2.2);
    Vector b(3.3, 4.4);

    // Now we get the usual output syntax
    cout << "vector + vector: a + b = " << a + b << endl;
    cout << "vector + scalar: b + 1.0 = " << b + 1.0 << endl;
    cout << "scalar + vector: 8.2 + a = " << 8.2 + a << endl;
}
```

friend Functions and friend Classes

- A class **X** may grant a function or another class as its friends.
- Friend functions are not considered member functions.
- Member access qualifiers are irrelevant to friend functions.
- Friend functions or classes of class **X** can be declared by **X** anywhere inside its class definition, but usually before all the members.
- Friends of **X** may access all its data members — both public and non-public members. So be careful!
- All member functions of an **X**'s friend class can access all data members of **X**.

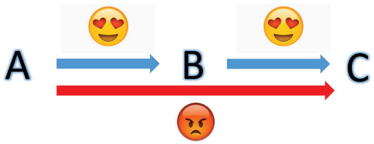
Properties of C++ Friendship

- Friendship is granted, not taken. The designer of a class determines who are its friends during the design.
  Afterwards, he cannot add more friends without rewriting the class definition.

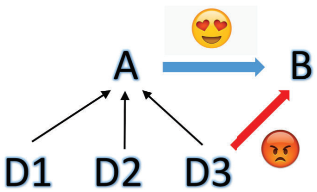


- Friendship is not symmetric: if A is B's friend, B is not necessarily A's friend.

Properties of C++ Friendship ..



- Friendship is not transitive: if A is B's friend and B is C's friend, A is not necessarily C's friend.



- Friendship is not inherited: friends of a base class do not become friends of its derived classes automatically.

Student with a Hacker Friend: v-student.h I

```
#ifndef V_STUDENT_H      /* File: v-student.h */
#define V_STUDENT_H
#include "course.h"
#include "v-uperson.h"

class Student : public UPerson
{
    // No forward declaration of class Hacker is needed
    friend class Hacker; // Got a Hacker friend! Good luck!

  private:
    float GPA; Course* enrolled[50]; int num_courses;

  public:
    Student(string n, Department d, float x) :
        UPerson(n, d), GPA(x), num_courses(0) { }

    ~Student()
        { for (int j = 0; j < num_courses; ++j) delete enrolled[j]; }
```

Student with a Hacker Friend: v-student.h II

```
    bool add_course(const string& s)
        { enrolled[num_courses++] = new Course(s); return true; };

    virtual void print() const
    {
        cout << "--- Student Details --- \n"
             << "Name: " << get_name()
             << "\nDept: " << get_department()
             << "\nGPA: " << GPA
             << "\n" << num_courses << " Enrolled courses: ";

        for (int j = 0; j < num_courses; ++j)
           { enrolled[j]->print(); cout << ' '; }

        cout << endl;
    }
};

#endif  // V_STUDENT_H
```


Student with a Bad Hacker Friend: hacker.h


```
#ifndef HACKER_H        /* File: hacker.h */
#define HACKER_H

class Hacker
{
  private:
    string name;

  public:
    Hacker(const string& s) : name(s) { }
    void add_course(Student& s) { s.GPA = 0.0; }   // Uh oh!!
};

#endif
```


Student with a Bad Hacker Friend: Ooops

```
#include <iostream>     /* File: bad-friend.cpp */
using namespace std;

#include "v-student.h"
#include "hacker.h"

int main()
{
    Student freshman("Naive", CIVL, 4.0);
    Hacker cool_guy("$#%&");

    freshman.print();
    freshman.add_course("COMP2012");
    freshman.print();

    cool_guy.add_course(freshman);
    freshman.print();
    return 0;
}
```

That's all!

Any questions?