Comp151

Generic Programming and
Standard Template Library (STL):

Introduction

Example: Class Person

```
class Person
{
  public:
    void set_name(const string& n);
    void set_address(const string& adr);
    void set_email_address(const string& adr);
    string get_email_address() const;
    string get_name() const;
    string get_address() const;

  private:
    string name;
    string address;
    string email_address;
};
```

Example: Class Person_Container As An Array

```
class Person_Container
{
  public:
    Person_Container(int n) : MAX_SIZE(n), size(0)
        { array = new Person [MAX_SIZE]; }
    int size() const { return size; }
    void add_person(const Person& pers);
    Person get_person(int i) const;
    Person delete_person(int i);

  private:
    const int MAX_SIZE;
    Person∗ array;               // One-time pre-allocated storage
    int size;                  // Number of Persons actually stored
};
```

Container Class

Classes that maintain collections of objects are so common that they have been given a name: Container Classes.

- Let's write a program to maintain a collection of persons, and apply some operations on that collection.
- The operations on `Person_Container` can be:
  - member functions of the `Person_Container` class.
  - global functions that take a `Person_Container&` argument.
- Here we print mailing labels for all the persons, and send emails to invite them to our party.
- However, in the future we may want to reuse the `Person_Container` in a completely different application.
- Thus, we'll keep the class interface small, and we make the functions that we only need in the current application global.

Example: Operations on Person_Container

```
void print_mailing_labels(const Person_Container& pc) {
    for (int i = 0; i < pc.size(); ++i) {
        Person pers = pc.get_person(i);
        cout << pers.get_name() << endl;
        cout << pers.get_address() << endl;
        // ...
    }
}

void invite_to_party(const Person_Container& pc) {
    for (int i = 0; i < pc.size(); ++i) {
        Person p = pc.get_person(i);
        string command = "cat party.txt | mail ";
        command += p.get_name();
        system( command.c_str() );              // Send invitation emails
    }
}
```

Similar Code

- Note the *similarities* in both functions: they both set up a loop to do something for all persons in the container.
- We can expect that if we add more functions that do something with all persons, that these functions show the same similarities.
- So we could reuse one of the existing functions by copying it, and changing the name and the body of for-loop of the copy. In fact, that is what we did when we made the second function.
- However, code reuse by copying is often a bad idea. Why?

Array or Linked List?

- In some applications it is very convenient that we implement Person_Container with an array; the get_person() member function takes only $O(1)$ (constant) time, and we use that member function a lot.
- However, in other applications we may find that we frequently need to merge two Person_Containers into a single one, or split one Person_Container into two Person_Container's.
- Now the fact that we use an array is a drawback (why?); a linked list would have been more practical in this case.
- So let's implement a container class called Person_List representing a list of Persons.

To Use a Linked List as a Container

The following interface functions are required:

- maintains a private pointer to the "current" element.
- `get_current( )` ⇒ get the current element.
- `get_first( )` ⇒ sets the pointer to the 1st item on the list.
- `get_next( )` ⇒ sets the pointer to the next element.
- `get_prev( )` ⇒ sets the pointer to the previous element.
- These functions return "-1" if there is nothing to point to.

We could, of course, add a member function `get_person(i)` that retrieves a person by index, but what would that do to the running time of `print_mailing_labels()`?

Example: print_mailing_labels() on Person_List

```
void print_mailing_labels(const Person_List& pl)
{
    if (pl.set_first() == -1)
        return;                                 // List is empty

    do
    {
        Person p = pl.get_current();
        cout ≪ p.get_name() ≪ endl;
        cout ≪ p.get_address() ≪ endl;
    } while (pl.set_next() != -1);      // End of list is reached
}
```

Example: invite_to_party() on Person_List

```
void invite_to_party(const Person_List& pl)
{
    if (pl.set_first() == -1)
        return;                                    // List is empty

    do
    {
        Person p = pl.get_current();
        string command = "cat party.txt | mail ";
        command += p.get_name();
        system( command.c_str() );     // Send invitation email
    } while (pl.set_next() != -1);     // End of list is reached
}
```

3 Concepts of Container Classes

In the previous examples, we can distinguish three concepts:

- the kind of container (list-based, array-based)
- the kind of objects stored in the container (`Person`, `int`)
- the kind of operations on the elements stored in the container ("do something for each element").

In our examples, there was a strong coupling between the three concepts:

- Whenever we change the type of the elements that we store, we have to re-implement the container (by copying and changing).
- Next, we also have to change the functions that deal with this container.

Similar Codes Again

Suppose that we want to search if a certain element is present in a container. Conceptually, the search algorithm is independent of the type of element: compare elements in the container with the search item, until you have found it, or no such element is present. However, in our examples, we would need separate functions for

- searching a `Person` with a specific name in a `Person_Container`;
- searching a `Person` with a specific name in a `Person_List`;

Now, if we work on integers instead, we would need to implement a `Int_Container` first; for instance, by copying and changing the `Person_Container`. Then, write separate functions for

- searching a specific value in a `Int_Container`;
- searching a specific value in a `Int_List`;

Generic Programming

- We see that strong coupling makes it impossible to reuse code without resorting to “code reuse by copying”. This leads to programs that are very inflexible, and difficult to maintain and extend. Isn't there a better way?
- It is possible to remove (or strongly reduce) the strong coupling between containers, contained elements, and operations on the elements of the container by applying Generic Programming.
- GP means “**programming with types as parameters**”.
- C++ supports GP through the template mechanism.

The Standard Template Library

- Containers are very common in programming, and several algorithms on container (searching for a specific element, sorting) occur in almost every non-trivial program.
- A lot of these general purpose containers can be found in the Standard Template Library, or STL for short.

To use the STL, we need an understanding of the following topics:

- templates
- operator overloading
- containers
- iterators
- function objects

Documentation for SGI's STL implementation can be found at: `http://www.sgi.com/tech/stl`.

Comp151

Generic Programming:

Function Templates & Class Templates

Example: max() by Copying

```
int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}

string max(const string& a, const string& b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Example: max() by Function Template

- A lot of times, we find a set of functions that look alike: e.g. for a certain type `T`, the `max` function has the form

  T max(**const** T& a, **const** T& b){ . . . }

- Instead of copying, in C++, we can just define one single function definition with templates:

```
template<typename T>
T max(const T& a, const T& b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Example: Use of Template Function max()

Now we can use `max()` for <u>any</u> type of arguments, as long as the arguments can be compared by " >":

```
int x = 4;
int y = 8;

cout ≪ max(x, y) ≪ " is a better number!" ≪ endl;


string a("Jesus");
string b("Buddha");

cout ≪ max(a, b) ≪ " is more powerful!  :-)" ≪ endl;
```

Function Template ..

The **typename** keyword may be replaced by **class**; the following template definition is equivalent to the one above:

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

The above template definitions are not functions; You cannot call a function template.

Function Template Instantiation

However, the compiler creates functions using function templates.

```
int i = 2, j = 3;
cout ≪ max(i, j);

string a("Hello"), b("World");
cout ≪ max(a, b);
```

- In this case, the compiler creates two `max()` functions using the function template

  ```
  template<typename T> T max(const T& a, const T& b);
  ```

- This is called template instantiation. The parameter `T` in the template definition is called the formal parameter or formal argument.
- In the above code, the template is instantiated with the actual arguments `int` and `string`, respectively.

Template: Formal Argument Matching

- When the compiler instantiates a template, it determines the actual type of the template parameter by looking at the types of the actual arguments:

```
template <typename T> T max(const T& a, const T& b);

void f() {
    cout ≪ max(3, 5);                                    // T is int;
    cout ≪ max(4.3, 5.6);                                // T is double
}
```

- However, there is <u>**NO**</u> automatic type conversion for template arguments:

```
cout << max(4, 5.5); // Error
```

- You can help by <u>*explicitly instantiating*</u> the function template:

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

Template: More Than One Formal Argument

Another way of using the `max` template with arguments of different types is changing its definition in the following way:

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

void f() {
    cout ≪ max(4, 5.5);                          // T1 is int, T2 is double
    cout ≪ max(5.5, 4);                          // T1 is double, T2 is int
}
```

However, there is a subtle problem here; the return type of `max` is the same as the type of the first argument. So what will the above code print?

Template: More Than One Formal Argument ..

The following template definition does not suffer from this problem:

```
template <typename T1, typename T2>
void print_max(const T1& a, const T2& b)
{
    cout ≪ ((a > b) ? a : b) ≪ endl;
}

void f()
{
    print_max(4, 5.5);                                    // Prints 5.5
    print_max(5.5, 4);                                    // Prints 5.5
}
```

Template Arguments: Too Many Combinations

Consider the following code:

```
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.
```

- The compiler should instantiate `print_max()` for 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, an innocent looking program may have a surprisingly large binary size, if you are not careful.

Class Templates: listnode.h

The template mechanism works for classes as well. This is particularly useful for defining container classes:

```
template <typename T>
class List_Node
{
  public:
    List_Node(const T& data);
    List_Node<T>* next();
    // Other member functions

  private:
    List_Node<T>* _next;
    List_Node<T>* _prev;
    T _data;

    friend class List<T>;
};
```

Class Templates: listnode.cpp

```
template <typename T>
List_Node<T>::List_Node(const T& data)
            : _data(data), _next(0), _prev(0) { }

template <typename T>
List_Node<T>* List_Node<T>::next() { return _next; }
```

Class Templates: list.h

Using the `List_Node<T>` class, we can define our list class template:

```
template <typename T>
class List
{
  public:
    List();
    void append(const T& item);
    // Other member functions

  private:
    List_Node<T>* _head;
    List_Node<T>* _tail;
};
```

Class Templates: list.cpp

```
template <typename T>
List<T>::List() : _head(0), _tail(0) { }

template <typename T>
void List<T>::append(const T& item)
{
    List_Node<T>* new_node = new List_Node<T>(item);
    if(!_tail)
        _head = _tail = new_node;
    else
        // ...
}
```

Class Templates: List Example

Now we can use our brand new list class to store any type of element that we want, without having to resort to “code reuse by copying”:

```
List<Person> people;
Person person("Brian");
people.append(person);
people.append(Person("Gary"));

List<int> primes;
primes.append(2);
primes.append(3);
primes.append(5);
```

Difference Between Class & Function Templates

Remember that for function templates, the compiler can deduce the template arguments:

```
int i = max(4, 5);
int j = max<int>(7, 2); // OK, but not needed
```

For class templates, you always have to specify the actual template arguments; the compiler does *not* deduce the template arguments:

```
List primes;      // Error
primes.append(2);
```

Function Template: Common Errors

```
template <class T> T* create() { ... } ;

template <class T> void f()
{
    T a;
    ...
}
```

e.g. With the template definition above, what is the function type of the following calls?

```
create();      // Error! Use e.g. create<int>(); instead
f();          // Error! Use e.g. f<float>(); instead
```

Reason: the compiler has to be able to deduce the actual function types from calls to the template function.

Separate Compilation For Templates??

For normal (non-template) functions, we usually put the declaration in a header file, and the definition in the corresponding *.cpp file. According to the C++ standard, we would expect this to work for function templates as well:

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

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

The same should apply to separating class definition in a `*.h` and implementation of class member functions in the corresponding *.cpp.

But a function/class is instantiated *only if* it is used *and* possibly on *every* use . . . ???

Can We Do This?

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

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

```
// File "a.cpp"
#include "max.h"
int main() { cout << max(2,4) << endl; }
```

```
g++ -c a.cpp; g++ -c max.cpp; g++ *.o
```

Inclusion Compilation of Templates

```
// File ”a.cpp”
#include ”max.h”

int main()
{
    cout ≪ max(2,4) ≪ endl;
}
```

```
// File ”max.h”

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

```
g++ -o a.out a.cpp
```

While there are other ways to compile templates, the simplest one is to include the template header file in every files which use the template.

Comp151

Generic Programming: Overloading Operator Functions

From Math Notation to Operators in Prog. Lang.[comp151]

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 like C only have operators defined for the *built-in* types.
- C++ is an exception: it allows you to redefine *most* of its operators for *user-defined* types. e.g. you may redefine "+", "-" etc. for types Complex, Matrix, Array, String, etc.

Example: Additions of Vectors

```
class Vector
{
    double _x, _y;
  public:
    Vector(double x, double y) : _x(x), _y(y) { }
    double x() const { return _x; }
    double y() const { return _y; }
};
```

- To add 2 vectors, traditionally we would do like this:

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

Vector a(1, 3), b(-5, 7), c(22, 2), d;
d = add(a, add(b, c));
```

Non-Member Operator Function

- It would be nicer if we could write the last expression

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

  as $d = a + b + c$.

- We can achieve that in C++ by simply replacing the name of the function `add` by `operator+`.

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

Vector a(1, 3), b(-5, 7), c(22, 2), d;
d = a + b + c;
```

Operator Syntax

- `operator+` is a formal function name that can be used like any other function name.
- Here we have used the “nickname”-syntax to call `operator+`. Instead, we could have used the “formal address” `operator+`:

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

  But who would want to write code like that?
- Operators in C++ are just like ordinary functions, except that they *also* have a nicer syntax for calling them similar to the usual mathematical notations.
- The operator + has a *formal name*, namely `operator+` (consisting of 2 keywords), and a “nickname,” namely +.

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 defining + to denote subtraction. There is nothing that inhibits you from defining `a = a + b` and `a += b` to have two different meanings. However, your code will become unreadable.

Don't shock the user!

C++ Operators

- Almost all operators in C++ can be overloaded except:

  `.`   `::`   `?:`   `sizeof`

- The C++ parser is fixed. That means that you can only *redefine existing operators*, but you CANNOT define *new* operators.
- Nor can you change the following properties of an operator:
  - 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 plus operator to take 3 arguments instead of 2.)
  - Associativity: e.g. `a+b+c` is always identical to `(a+b)+c`.
  - Precedence: which operator is done first?
    e.g. `a+b*c` is treated as `a+(b*c)`.

C++ Operators . . .

- All C++ operators already have predefined meaning for the built-in types. It is impossible to change this meaning; you can only *overload* the operator to have a meaning for your *own* (user-defined) classes (such as `Vector` in the example above).
- Therefore, every operator you define must have *at least one* argument of a user-defined class type.
- As a global function, `operator+` has two arguments. When it is called in an expression such as `a + b`, this is equivalent to writing `operator+(a, b)`.

Member Operator Function

Member functions are called using the “dot syntax” by specifying an object of, for example, type `Vector`.

- The expression `a + b` is equivalent to `a.operator+(b)`.
- 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.

```
class Vector {
    double _x, _y;
  public:
    Vector(double x, double y) : _x(x), _y(y) { }
    double x() const { return _x; }
    double y() const { return _y; }
    Vector operator+ (const Vector& b) const
        { return Vector( _x + b._x, _y + b._y ); }
};
```

Member and Non-Member Operator Function

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

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

and verifies whether one of those two operator functions are defined.

Note: It is an ERROR to define both.

Example: Member or Non-Member Function?

Let's define a multiplication operator to multiply a vector with a scalar. This should all work:

```
Vector a(1,0), b(2, 3);
Vector c = 2 * a;                 // c == (2, 0)
a = c + b * 3;                    // a == (8, 9)
```

Can we define the multiplication operator as a member function of `Vector`?

- Remember that the compiler converts the expression `a*b` to `a.operator*(b)`. So the expression `2*a` is converted to `2.operator*(a)`!

Example: Member or Non-Member Function? ...

- This doesn't work! 2 is an object of type `int`, and we cannot define a new member function for this type.
- So our only choice is to define the multiplication operator as a global non-member function:

```
Vector operator* (double s, const Vector& a)
{
    return Vector(s * a.x(), s * a.y());
}
```

Example: Operator Function for Printing

Very often you would like to provide a printing service for your user-defined classes, and the most natural way of doing that is to define the << operator for your class.

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

- `ostream` is the base class for all possible output streams.
- In particular, the standard output stream `cout` and the error output stream `cerr` are objects of classes derived from `ostream`.

Example: Operator Function for Printing ..

Why does the operator return an output stream?

Because we like to write expressions 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 output stream itself.

Quiz: Could we have defined `operator<<` as a member function?

Operator: Member or Non-Member 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. ⇒ 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<<`
- To allow automatic conversion of types using the conversion constructor, for commutative operators like “+”, “-”, “*”, it is usually preferred to be defined as non-member functions. e.g.

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

How to Differentiate Prefix and Postfix Operators?[comp151]

```
class Vector {
      . . .
   public:
      Vector() : _x(0.0), _y(0.0) { }
      Vector(double x, double y) : _x(x), _y(y) { }
      Vector operator++() { ++_x; ++_y; return *this; }
      Vector operator++(int)
            { Vector temp(_x,_y); _x++; _y++; return temp; }
};

int main() {
      Vector a(1.2, 3.4), c, d;
      c = ++a;                                  // c =  2.2 + 4.4 j
      d = a++;                                  // d =  1.2 + 3.4 j
}
```

Comp151

Generic Programming:

Container Classes

Container Classes

- Container classes are a typical use for class templates, since we need container classes for objects of many different types, and the types are not known when the container class is designed.
- Let's design a container that looks like an array, but that is a first-class type: so assignment and call by value is possible.
- We want the container to be *homogeneous*: all the elements must have the same type.
- But should a container with 10 `int` elements be the same type as a container with 20 `int` elements?

  Both are sensible design decisions.

Remark: The `vector` type in STL is better, so this is just for our understanding.

Example: Container Class — bunch.h

```
template <typename T, int N>
class Bunch
{
  private:
   T _value[N];

  public:
   int size() const { return N; }
   T& operator[ ](int i) { return _value[i]; }
   . . .
};
```

Example: Use of Class Bunch

```
Bunch<int, 10> A;
cout << A[3];
A[7] = 13;
A[2]++;

Bunch<string, 50> B;
B[49] = "Hello world";

Bunch<string, 50> C;
C = B;                                          // Legal

Bunch<int, 20> D;
D = A;                   // Error: D and A are of different types
```

A More Flexible Container Class — array.h

```
#ifndef ARRAY_H
#define ARRAY_H

template <typename T>
class Array
{
  private:
    T* _value;
    int _size;
  public:
    Array(int n = 10);                        // Default or conversion constructor ??
    Array(const Array<T>& a);                                   // Copy constructor
    ~Array();

    int size() const { return _size; }
    void init(const T& k);
    Array<T>& operator=(const Array<T> &a);                  // Assignment operator
    T& operator[ ](int i) { return _value[i]; }              // Access to an element
    const T& operator[ ](int i) const { return _value[i]; }  // Inspect an element
};

#endif                                                                // ARRAY_H
```

Example: Use of Class Array

```
#include <iostream>
#include ”array.h”

int main()
{
    Array<int> a(3); a.init(8); cout ≪ a;
    Array<int> b = 5; b.init(99); cout ≪ b;

    a = a; a[2] = 17; cout ≪ a;
    a = b; cout ≪ a;

    const Array<int> c(20);
    c[1] = 5;                    // Error: assignment of read-only location
    cout ≪ c[1] ≪ endl;

    return 0;
}
```

Example: Constructors/Destructor of Class Array

```
template <typename T>
Array<T>::Array(int n) : _value( new T[n] ), _size (n) { }

template <typename T>
Array<T>::Array(const Array<T> &a)
     : _value( new T[a._size] ), _size (a._size)
{
     for (int i = 0; i < _size; ++i) _value[i] = a._value[i];
}

template <typename T>
Array<T>::~Array() { delete [ ] _value; _value = 0; _size = 0; }

template <typename T>
void Array<T>::init(const T& k)
{
     for (int i = 0; i < _size; ++i) _value[i] = k;
}
```

Shallow Copy and Deep Copy

```
Array<int> A(10);
Array<int> B(A);
```

Shallow Copy:

- If you don't define your own copy constructor, the copy constructor provided by the compiler simply does member-wise copy.
- Then A and B will share to the same `_value` array.
- If you delete A, and then B, you will get error as you will delete the embedded `_value` array twice from the heap.
- Basically, shallow copy is a bad idea if an object *owns* data.

Deep Copy:

- To take care of the ownership, redefine the copy constructor so that each object has its own copy of the “owned” data members.

Assignment Operator

Idea: To assign `b = a`, first throw away the old data `b._value`, then create a new one and assign the elements from `a._value`.

```
template <typename T>
Array<T>& Array<T>::operator=(const Array<T>& x)
{
    delete [ ] _value;                          // Destroy own memory

    // Re-allocate memory of the right size
    _size = x._size;
    _value = new T[_size];

    // Copy over
    for (int i = 0; i < _size; ++i)
        _value[i] = x._value[i];

    return *this;
}
```

Assignment Operator ..

Problem: What if you do: a = a?

Solution: When the assignment argument is the same as the object being assigned to, don't do anything.

```
template <typename T>
Array<T>& Array<T>::operator=(const Array<T>& x)
{
    if (&x != this)
    {
        this→~Array();                              // Destroy own memory

        _size = x.size();                           // Re-allocate memory
        _value = new T [_size];

        for (int j=0; j <_size; ++j) _value[j] = x[j];        // Copy over
    }
    return (*this);
}
```

Assignment Operator ...

Here is another way of implementing the assignment operator.

```
#include <stl.h>
template <typename T>
Array<T>& Array<T>::operator=(const Array<T>& A)
{
      _size = A._size;
      Array<T> temp(A);
      std::swap(_value, temp._value);
      return *this;
}

template <typename T>
void swap(T& a, T& b)
{
      T temp = a;
      a = b;
      b = temp;
}
```

Output Operator as Global Function

- The following output operator is not a member of the `Array<T>` class, but a function template.
- Function templates and class templates work together very well: We can use function templates to implement functions that will work on any class created from a class template.

```
template <typename T>
ostream& operator≪(ostream& os, const Array<T>& x)
{
    os ≪ "#elements stored = " ≪ x.size() ≪ endl;
    for (int i = 0; i < x.size(); ++i)
        os ≪ x[i] ≪ endl;

    return os;
}
```

Output Operator as Friend Function

```
template <typename T>
class Array
{
    friend ostream& operator≪(ostream& os, const Array<T>& x)
    {
        os ≪ "#elements stored = " ≪ x._size ≪ endl;
        for (int i = 0; i < x._size; ++i)
            os ≪ x._value[i] ≪ endl;
        return os;
    }

  private:
    T∗ _value; int _size;
  public: ...
};
```

Output Operator as Friend Function: only in g++

```
template <typename T>
class Array
{
    friend ostream& operator≪ <T>(ostream&, const Array<T>&);
  private:
    T∗ _value; int _size;
  public: . . .
};

template <typename T>
ostream& operator≪(ostream& os, const Array<T>& x);
{
    os ≪ "#elements stored = " ≪ x._size ≪ endl;
    for (int i = 0; i < x._size; ++i)
        os ≪ x._value[i] ≪ endl;
    return os;
}
```

Why 2 Different Subscript Operators?

We have 2 subscript operators, and it looks as if we are violating the overloading rule. Both have the same name and the same arguments.

```
Array<int> a(3);
a[2] = 7;
```

In the above code, we need a subscript operator that returns a "`int &`", not a "`const int &`".

But this subscript operator does not work in this code:

```
int last_element(const Array<int>& a)
{
  return a[a.size() - 1];
}
```

Why 2 Different Subscript Operators? ..

The argument "a" of `last_element( )` is a "`const Array<int> &`". Therefore it can *only* call `const` member functions: in this example,

- `int size() const`
- `const T& operator[ ](int i) const`

On the other hand, if you are not so strict with const-ness, you may simply define one subscript function as:

```
T& operator[ ](int i) const { return _value[i]; }
```

Comp151

Generic Programming:
Iterators

Standard Sequence Containers

Here are some homogeneous container classes commonly used to represent a sequence of objects of the same type.

Container	Access Control	Add/Remove
vector (1D array)	O(1) random access	O(1) at the end O(n) in front/middle
list (doubly-linked list)	O(n) in middle O(1) at front/end	O(1) at any position
deque (doubly-ended queue)	O(1) random access	O(1) at front/back O(n) in middle

Sequence Container: Access, Add, Remove

Element access for all:

- `front()`: First element
- `back()`: Last element

Element access for `vector` and `deque`:

- `[ ]`: Subscript operator, index not checked.

Add/remove elements for all:

- `push_back()`: Append element.
- `pop_back()`: Remove last element.

Add/remove elements for `list` and `deque`:

- `push_front()`: Insert element at the front.
- `pop_front()`: Remove first element.

Sequence Container: Other Operations

"List" operations are fast for `list`, but also available for `vector` and `deque`:

- `insert(p, x)`: Insert an element at a given position.
- `erase(p)`: Remove an element.
- `clear()`: Erase all elements.

Miscellaneous Operations:

- `size()`: Returns the number of elements.
- `empty()`: Returns true if the sequence is empty.
- `resize(int i)`: Change size of the sequence.

Comparison operators ==, !=, < etc. are also defined.

Example: Print with an Array

```
const int LEN = 10;
int x[LEN];
int* const x_end = &x[LEN];

for (int* p = x; p != x_end; ++p)
    cout << *p;
```

We use an `int` pointer to access the elements of an `int` sequence with some basic operations:

Operation	Goal
`p = x`	initialize to the beginning of an array
`*p`	access the element being pointed to
`++p`	point to the next element
`p != x_end`	compare with the end of an array

Example: Print with a List

To access the list elements *sequentially*, one may define an iterator `list<int>::iterator`, and add functions `begin()` and `end()` to set to the beginning and end of the container.

```
#include <list>                              // "list" class of STL

list<int> x;
list<int>::iterator p;

for (p = x.begin(); p != x.end(); ++p)
    cout << *p;
```

Iterators

- For each kind of container in the STL there is an *iterator type*.

```
list<int>::iterator lp;
vector<string>::iterator vp;
deque<double>::iterator dp;
```

- Iterators are used much like pointers: They can be used to indicate elements in the sequence, or to indicate a subsequence.
- Operations on iterators are:
  - Access element: `*p`, `p->`
  - Go to next or previous element: `++p`, `--p`
  - Compare iterators: `==`, `!=`

Example: find() with an int Iterator

- Iterator provides a systematic way to look at elements of sequence container without making any difference between different container classes.
- The same code works correctly for all sequence container classes.

```
// File: "find_int_iterator.cpp"
typedef int* Int_Iterator;

Int_Iterator find( Int_Iterator begin, Int_Iterator end, const int& value )
{
    while (begin != end && *begin != value)
        ++begin;

    return begin;
}
```

Example: find() with an int Iterator . . .

```
#include "find_int_iterator.cpp"

int main() {
    const int SIZE = 100; int x[SIZE]; int num;
    Int_Iterator begin = x; Int_Iterator end = &x[SIZE];

    for (int i = 0; i < SIZE; i++)
        x[i] = 2 ∗ i;

    while (true) {
        cout ≪ "Enter number:   "; cin ≫ num;
        Int_Iterator position = find(begin, end, num);

        if (position != end) {
            ++position;
            if (position != end) cout ≪ "Found before " ≪ ∗position ≪ '\n';
            else cout ≪ "Found as last element\n";
        }
        else cout ≪ "Not found\n";
    }
}
```

Why Are Iterators So Great?

Because they allow us to separate algorithms from containers.

- If we change the `find()` function as follows, it still works:

```
template <class Iterator, class T>
Iterator find( Iterator begin, Iterator end, const T & value )
{
    while (begin != end && *begin != value)
        ++begin;
    return begin;
}
```

- The new `find()` function contains no information about the implementation of the container, or how to move the iterator from one element to the next.
- The same `find()` function can be used for any container that provides a suitable iterator.

Example: find() with an Iterator

```
#include <vector>                                             // "vector" class from STL
int main()
{
    vector<int> x(SIZE); int num;
    for (int i = 0; i < SIZE; i++)
        x[i] = 2 * i;

    while (true)
    {
        cout ≪ "Enter number:   "; cin ≫ num;
        vector<int>::iterator position = find(x.begin(), x.end(), num);

        if (position != x.end())
        {
            ++position;
            if (position != x.end()) cout ≪ "Found before " ≪ *position ≪ '\n';
            else cout ≪ "Found as last element.\n";
        }
        else cout ≪ "Not found\n";
    }
}
```

Comp151

STL: Algorithms

Example: STL Algorithm — find()

The Standard Template Library not only contains container classes, but also algorithms that operate on sequence containers:

```
#include <algorithm>
#include <string>
#include <list>

int main()
{
    list<string> composer;
    composer.push_back("Mozart"); composer.push_back("Bach");
    composer.push_back("Chopin"); composer.push_back("Beethoven");

    list<string>::iterator p = find(composer.begin(), composer.end(), "Bach");
    if (p == composer.end())
        cout ≪ "Not found." ≪ endl;
    else if(++p != composer.end())
        cout ≪ "Found before:  " ≪ *p ≪ endl;
    else
        cout "Found at the end." ≪ endl;
}
```

Algorithms, Iterators, and Sub-Sequences

Sequences/Sub-sequences are specified using iterators that indicate the beginning and the end for an algorithm to work on.

Here we find the *2nd* occurrence of the value, 341, in a sequence.

```
// File "init.cpp"
inline int f(int x) { return -x*x + 40*x + 22; }

template <typename T>
void my_initialization(T& x)
// or, void my_initialization(vector<T>& x)
{
    const int N = 10;
    for (int j = 0; j < N; ++j)
        x.push_back( f(j) );
}
```

Example: Algorithm with Iterators & Sub-Sequence

```
#include <vector>
#include <algorithm>
#include ”init.cpp”

int main()
{
    const int search_value = 341;
    vector<int> x; my_initialization(x);

    vector<int>::iterator p = find(x.begin(), x.end(), search_value);

    if (p != x.end())                                   // Value found!
    {
        p = find(++p, x.end(), search_value);           // Find again
        if (p != x.end())                         // Value found again!
            cout ≪ "Found after:  " ≪ *--p ≪ endl;
    }
}
```

STL Algorithms — find()

```
template <class Iterator, class T>
Iterator find(Iterator first, Iterator last, const T& value)
{
    while (first != last && *first != value)
        ++first;
    return first;
}
```

- `find()` searches linearly through a sequence, and stops when an item matches the 3rd argument.
- A big limitation of `find()` is that it requires an *exact* match by value.

Generic Algorithms with Function Arguments

Let's search a value in a container which satisfies a boolean condition that is specified by a C++ function.

```
#include <vector>
#include <algorithm>
#include "init.cpp"

bool greater_than_350(int value) { return value > 350; }

int main()
{
    vector<int> x; my_initialization(x);
    vector<int>::iterator p = find_if( x.begin(), x.end(), greater_than_350 );

    if (p != x.end())
        cout << "Found element:  " << *p << endl;
}
```

STL Algorithms — find_if()

```
template <class Iterator, class Predicate>
Iterator find_if(Iterator first, Iterator last, Predicate pred)
{
    while (first != last && !pred(*first))
        ++first;
    return first;
}
```

- `find_if()` is a more general algorithm than `find()` in that it stops when a condition is satisfied.
- This allows partial match, or match by keys.
- The condition is specified by a function.

C Function Pointer

- Inherited from C, C++ allows a function to be passed as argument to another function.
- Actually, we say that we pass the function pointer.
- e.g. if you do "`man 3 qsort`" on Unix:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compare)(const void *, const void *))
```

i.e. The 4th argument is a function pointer, whose type is:

```
    int (*)(const void*, const void*)
```

e.g. `int compare(int* i, int* j) { return (*i) - (*j); }`

- Similarly, the type for the template `max()` function pointer we talked before is:

```
    T (*)(const T&, const T&)
```

Example: C Function Pointer

```
#include <iostream>

int max(int x, int y) { return (x > y) ? x : y; }
int min(int x, int y) { return (x > y) ? y : x; }

void main()
{
    int (*f)(int x, int y);
    int choice;

    cin ≫ choice;
    if (choice == 1)
        f = max;
    else
        f = min;

    cout ≪ f(3,5) ≪ endl;
}
```

Function Objects

STL has a more generalized concept of function pointer: Any “object” that can be “called” is a function objects, and function pointer is just one example.

- An object can be called if it supports the `operator()`.

```
// File "greater_than.h"
class Greater_Than
{
  private:
    int limit;
  public:
    Greater_Than(int a) : limit(a) { }
    bool operator()(int value) { return value > limit; }
};
```

How to Use Function Objects in STL Algorithms?

```
#include <vector>
#include <algorithm>
#include ”greater_than.h”
#include ”init.cpp”

int main()
{
    vector<int> x; my_initialization(x);
    vector<int>::iterator p =
        find_if( x.begin(), x.end(), Greater_Than(350) ); // <—

    if (p != x.end())
        cout ≪ "Found element " ≪ *p ≪ endl;
}
```

How to Use Function Objects STL Algorithms? . . .

- The line <-- is the same as:

```
Greater_Than g(350);
// Create a Greater_Than function object
p = find_if( x.begin(), x.end(), g );
```

- When `find_if()` examines each item, say `x[j]` in the container `vector<int> x`, the `Greater_Than` function object will be called using its `operator()` with the container item. i.e.

```
g(x[j]) // Or in formal writing: g.operator()(x[j])
```

Function objects . . .

An object that can be called like a function is called a function object, functoid, or functor.

- Function objects are more powerful than functions, since they can have data members and therefore carry around information or *internal states*.
- A function object must have at least the operator() overloaded so that it can be called.
- A function object (or a function) that returns a boolean value (of type `bool`) is called a predicate.

Example: STL Algorithm — for_each() to Sum

```
#include <list>
#include <algorithm>
#include "init.cpp"

class Sum {
  private:
    int sum;
  public:
    Sum() : sum(0) { }
    void operator()(int value) { sum += value; }
    int result() const { return sum; }
};

int main()
{
    list<int> x; my_initialization(x);
    Sum sum = for_each( x.begin(), x.end(), Sum() );
    cout << "Sum = " << sum.result() << endl;
}
```

STL Algorithms — for_each(), transform()

```
template <class Iterator, class Function>
Function for_each(Iterator first, Iterator last, Function g)
{
    for ( ; first != last; ++first)
        g(*first);
    return g;
}


template <class Iterator1, class Iterator2, class Function>
Iterator2 transform(Iterator1 first, Iterator1 last, Iterator2 result, Function g)
{
    for ( ; first != last; ++first, ++result)
        *result = g(*first);
    return result;
}
```

Example: STL Algorithm — count_if()

Here we count the number of elements that are larger than 10:

```
#include <vector>
#include <algorithm>
#include "greater_than.h"
#include "init.cpp"

int main()
{
    vector<int> x;
    my_initialization(x);
    int num = count_if( x.begin(), x.end(), Greater_Than(10) );
}
```

Example: STL Algorithm — for_each() to Add

```
// File: "add.h"
#include <list>
#include <vector>
#include <algorithm>

class Add {
  private: int data;
  public:
    Add(int i) : data(i) { }
    int operator()(int value) { return value + data; }
};

class Print {
  private: ostream& os;
  public:
    Print(ostream& s) : os(s) { }
    void operator()(int value) { os ≪ value ≪ "  "; }
};
```

Example: STL Algorithm — for_each() to Add . . .

```
#include "add.h"

int main()
{
    list<int> x; my_initialization(x);
    vector<int> y(x.size());

    transform( x.begin(), x.end(), y.begin(), Add(10) );
    for_each( y.begin(), y.end(), Print(cout) );
    cout ≪ endl;
}
```

Example: STL Algorithm — transform() and copy()

The `istream_iterator` and `ostream_iterator` templates make it possible to treat streams as sequences:

```
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>
#include "add.h"
#include "init.cpp"

int main()
{
    list<int> x; my_initialization(x);
    vector<int> y;

    transform(x.begin(), x.end(), back_inserter(y), Add(77) );
    copy(y.begin(), y.end(), ostream_iterator<int>( cout, "\n") );
}
```

Other Algorithms in the STL

- min_element and max_element
- equal
- generate (Replace elements by applying a function object)
- remove, remove_if Remove elements
- reverse, rotate Rearrange sequence
- random_shuffle
- binary_search
- sort (using a function object to compare two elements)
- merge, unique
- set_union, set_intersection, set_difference

Documentation for SGI's STL implementation can be found at: http://www.sgi.com/tech/stl.