

C++ Object-Oriented Prog.

Unit 6: Generic Programming

CHAPTER 23: GENERIC ALGORITHMS AND OPERATOR OVERLOADING

DR. ERIC CHOU

IEEE SENIOR MEMBER

LECTURE 1

Generic Algorithm and Operator Overloading



Generic Algorithms

- **Generic algorithms** are algorithms in which the actions or steps are defined, but the data types of the items being manipulated are not

Example of a Generic Algorithm

```
void PrintInt(int n){  
    cout << "***Debug" << endl;  
    cout << "Value is " << n << endl;  
}  
  
void PrintChar(char ch){  
    cout << "***Debug" << endl;  
    cout << "Value is " << ch << endl;  
}  
  
void PrintFloat(float x){  
}  
  
void PrintDouble(double d){  
}
```

To output the traced values, we insert:

```
sum = alpha + beta + gamma;  
PrintInt(sum);  
  
PrintChar(initial);  
  
PrintFloat(angle);
```



Function Overloading

- **Function overloading** is the use of the same name for different functions, distinguished by their parameter lists
- Eliminates need to come up with many different names for identical tasks
- Reduces the chance of unexpected results caused by using the wrong function name

Example of Function Overloading

```
void Print(int n)
{
    cout << "***Debug" << endl;
    cout << "Value is " << n << endl;
}
void Print(char ch)
{
    cout << "***Debug" << endl;
    cout << "Value is " << ch << endl;
}
void Print(float x)
{
}
```

To output the traced values, we insert:

```
Print(someInt);
Print(someChar);
Print(someFloat);
```



Function Template

- A C++ language construct that allows the compiler to generate multiple versions of a function by allowing parameterized data types

FunctionTemplate

```
Template < TemplateParamList >  
FunctionDefinition
```

TemplateParamDeclaration

```
{ class  
  Identifier  
  typename
```

Example of a Function Template

```
template<class SomeType>
```

*Template
parameter*

```
void Print(SomeType val)
```

```
{
```

```
    cout << "***Debug" << endl;
```

```
    cout << "Value is " << val << endl;
```

```
}
```

*Template
argument*

To output the traced values, we insert:

```
Print<int>(sum);
```

```
Print<char>(initial);
```

```
Print<float>(angle);
```




Instantiating a Function Template

- When the compiler instantiates a template, it substitutes the template argument for the template parameter throughout the function template

TemplateFunction Call

Function < TemplateArgList > (FunctionArgList)



Implicit template arguments

`Print(sum);` `// Implicit: Print<int>(sum)`

`Print(initial);` `// Implicit: Print<char>(initial)`

`Print(angle);` `// Implicit: Print<float>(angle)`



Enhancing the Print Template

```
template<class SomeType>
void Print( string vName,
           // Name of the variable
           SomeType val )
           // Value of the variable
{
    cout << "***Debug" << endl;
    cout << "Value of " << vName
         << " = " << val << endl;
}
```



Generic Functions, Function Overloading, Template Functions

Generic Function

Different Function Definitions
Different Function Names

Function Overloading

Different Function Definitions
Same Function Name

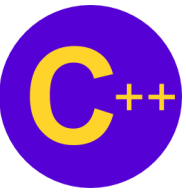
Template Functions

One Function Definition (a function template)
Compiler Generates Individual Functions

User-Defined Specializations

Example that demonstrates use of template < >

```
template<>
void Print(string      vName,      // Name of the variable
           StatusType val )      // Value of the variable
{
    cout << "***Debug" << endl;
    cout << "Value of " << vName << " = ";
    switch (val) {
        case OK : cout << "OK";
                   break;
        case OUT_OF_STOCK : cout << "OUT_OF_STOCK";
                             break;
        case BACK_ORDERED : cout << "BACK_ORDERED";
                             break;
        default : cout << "Invalid value";
    }
    cout << endl;
}
```



Organization of Program Code

Three possibilities:

1. Template definitions at the beginning of program file, prior to `main` function
2. Function prototypes first, then the `main` function, then the template definitions
3. Template definition in the header file, use `#include` to insert that file into the program

LECTURE 1

Generic Data Types



What is a Generic Data Type?

- It is a type for which the operations are defined but the data types of the items being manipulated are not

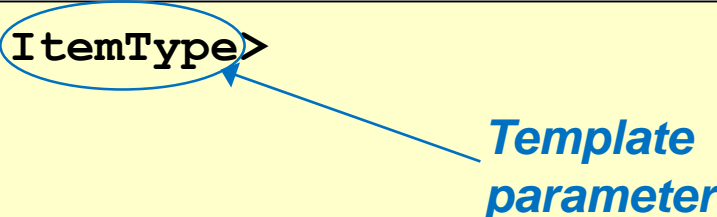


What is a Class Template?

- It is a C++ language construct that allows the compiler to generate multiple versions of a class by allowing parameterized data types

Example of a Class Template

```
template<class ItemType>
class GList
{
public:
    bool IsEmpty() const;
    bool IsFull() const;
    int Length() const;
    void Insert(/* in */ ItemType item);
    void Delete(/* in */ ItemType item);
    bool IsPresent(/* in */ ItemType item) const;
    void SelSort();
    void Reset() const;
    ItemType GetNextItem();
    GList(); // Constructor
private:
    int length;
    ItemType data[MAX_LENGTH];
};
```





Instantiating a Class Template

To create lists of different data types

```
// Client code
```

```
GList<int> list1;  
GList<float> list2;  
GList<string> list3;
```

```
list1.Insert(356);  
list2.Insert(84.375);  
list3.Insert("Muffler bolt");
```

template argument



Compiler generates 3
distinct class types

```
GList_int list1;  
GList_float list2;  
GList_string list3;
```



Instantiating a Class Template

- Class template arguments *must* be explicit
- The compiler generates distinct class types called **template classes** or generated classes
- When instantiating a template, a compiler substitutes the template argument for the template parameter throughout the class template

Substitution Example

```
class GList_int
{
public:

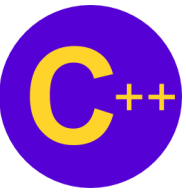
void Insert(/* in */ ItemType item);

void Delete(/* in */ ItemType item);

bool IsPresent(/* in */ ItemType item) const;

private:
    int length;
    ItemType data[MAX_LENGTH];
};
```

The diagram illustrates the substitution of the type `int` for the abstract type `ItemType` in the `GList_int` class. Four blue arrows point from the word `int` to the four occurrences of `ItemType` in the code: the parameter of `Insert`, the parameter of `Delete`, the parameter of `IsPresent`, and the array element type in `data`.



Writing Function Templates

```
template<class ItemType>
void GList<ItemType>::Insert(/* in */ ItemType item)
{
    data[length] = item;
    length++;
}

void GList<float>::Insert(/* in */ float item)
{
    data[length] = item;
    length++;
}
```



Organization of Program Code

A compiler must know the argument to the template in order to generate a function template, and this argument is located in the client code

Solutions

- Have specification file include implementation file
- Combine specification file and implementation file into one file



Warning!

- Are you using an IDE (integrated development environment) where the editor, compiler, and linker are bundled into one application?
- **Remember** The compiler must know the template argument
- How you organize the code in a project may differ depending on the IDE you are using

LECTURE 1

Operator Overloading



Overview for Operator Overloading

- Use operators with objects (operator overloading)
 - Clearer than function calls for certain classes
 - Operator sensitive to context
- Examples
 - <<
 - Stream insertion, bitwise left-shift
 - +
 - Performs arithmetic on multiple items (integers, floats, etc.)



Fundamentals of Operator Overloading

- Types for operator overloading
 - Built in (int, char) or user-defined (classes)
 - Can use existing operators with user-defined types
 - Cannot create new operators
- Overloading operators
 - Create a function for the class
 - Name of operator function
 - Keyword operator followed by **symbol**
 - Example
 - operator+ for the addition operator +



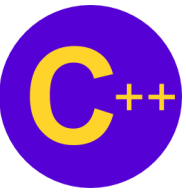
Fundamentals of Operator Overloading

- Using operators on a class object
 - It must be overloaded for that class
 - Exceptions: (can also be overloaded by the programmer)
 - Assignment operator (=)
 - Memberwise assignment between objects
 - Address operator (&)
 - Returns address of object
 - Comma operator (,)
 - Evaluates expression to its left then the expression to its right
 - Returns the value of the expression to its right
- Overloading provides concise notation
 - `object2 = object1.add(object2);`
vs.
`object2 = object2 + object1;`



Restrictions on Operator Overloading

- Cannot change
 - Precedence of operator (order of evaluation)
 - Use parentheses to force order of operators
 - Associativity (left-to-right or right-to-left)
 - Number of operands
 - e.g., & is unary, can only act on one operand
 - How operators act on built-in data types (i.e., cannot change integer addition)
- Cannot create new operators
- **Operators must be overloaded explicitly**
 - Overloading + and = does not overload +=
- Operator ? : cannot be overloaded



Operators that can/cannot be overloaded.

Operators that can be overloaded

| | | | | | | | |
|--------------|-----------------|----|----|----|----|------------|---------------|
| + | - | * | / | % | ^ | & | |
| ~ | ! | = | < | > | += | -= | *= |
| /= | %= | ^= | &= | = | << | >> | >>= |
| <<= | == | != | <= | >= | && | | ++ |
| -- | ->* | , | -> | [] | () | new | delete |
| new[] | delete[] | | | | | | |

Operators that cannot be overloaded

| | | | |
|---|----|----|----|
| . | .* | :: | ?: |
|---|----|----|----|



Software Engineering Observation

- At least **one** argument of an operator function must be an **object** or **reference** of a **user-defined** type.
- This prevents programmers from changing how operators work on fundamental types.
- Assuming that overloading an operator such as `+` overloads related operators such as `+=` or that overloading `==` overloads a related operator like `!=` can lead to **errors**.
- Operators can be overloaded only explicitly; **there is no implicit overloading**.

LECTURE 1

Operator Class Member of Non- Member



Operator Functions as Class Members vs. Global Members

- **Operator functions**

- **As member functions**

- **Leftmost** object must be of same class as operator function
 - Use this keyword to implicitly get left operand argument
 - Operators (), [], -> or any assignment operator must be overloaded as a class member function
 - Called when
 - Left operand of binary operator is of this class
 - Single operand of unary operator is of this class

- **As global functions**

- Need parameters for both operands
 - Can have object of different class than operator
 - Can be a **friend** to access **private** or **protected** data



Operator Functions as Class Members vs. Global Members

- Overloaded << operator
 - Left operand of type **ostream &**
 - Such as cout object in cout << classObject
 - Similarly, overloaded >> has left operand of istream &
 - Thus, both must be **global** functions
- << and >> operators
 - Already overloaded to process each built-in type
 - Can also process a user-defined class
 - Overload using global, friend functions



Operator Functions as Class Members vs. Global Members

- Commutative operators
 - May want + to be commutative
 - So both “a + b” and “b + a” work
- Suppose we have two different classes
 - Overloaded operator can **only** be member function when its class is on left
 - HugeIntClass + long int
 - Can be member function
 - When other way, need a **global** overloaded function
 - long int + HugeIntClass



Overloading Unary Operators

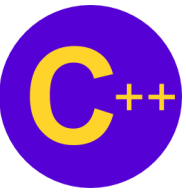
- Overloading unary operators
 - Can overload as non-static member function with no arguments
 - Can overload as global function with one argument
 - Argument must be class object or reference to class object
 - Remember, static functions only access static data

Overloading Unary Operators

- Example:
 - Overload ! to test for empty string
 - If non-static member function, needs no arguments

```
class String
{
    public:
        bool operator!() const;
    ...
};
```

- !s becomes s.operator!()
- If **global** function, needs one argument
 - `bool operator!(const String &)`
 - !s becomes operator!(s)



Overloading Binary Operators

- Overloading binary operators
 - Non-static member function, one argument
 - Global function, two arguments
 - One argument must be class object or reference



Overloading Binary Operators

- Example: Overloading +=
 - If non-static member function, needs one argument

```
class String
{
public:
    const String & operator+=( const String & );
    ...
};
```
 - If global function, neey += z becomes y.operator+=(z)
 - ds two arguments
 - `const String & operator+=(String &, const String &);`
 - `y += z` becomes `operator+=(y, z)`

LECTURE 1

Case Study: Array Class



Case Study: Array Class

- Pointer-based arrays in C++
 - No range checking
 - Cannot be compared meaningfully with `==`
 - No array assignment (array names are `const` pointers)
 - If array passed to a function, size must be passed as a separate argument
- Example: Implement an `Array` class with
 - Range checking
 - Array assignment
 - Arrays that know their own size
 - Outputting/inputting entire arrays with `<<` and `>>`
 - Array comparisons with `==` and `!=`

Case Study: Array Class

- Copy constructor
 - Used whenever copy of object is needed:
 - Passing by value (return value or parameter)
 - Initializing an object with a copy of another of same type
 - `Array newArray(oldArray);` or
`Array newArray = oldArray` (both are identical)
 - `newArray` is a copy of `oldArray`
 - Prototype for class Array
 - `Array(const Array &);`
 - Must take reference
 - Otherwise, the argument will be passed by value...
 - Which tries to make copy by calling copy constructor...
 - Infinite loop

```
1 #ifndef ARRAY_H
2 #define ARRAY_H
3 #include <iostream>
4 #include <fstream>
5 using namespace std;
6
7 class Array{
8     friend ostream &operator<<(ostream&, const Array&);
9     friend istream &operator>>(istream&, Array&);
10 public:
11     Array(int=10);           // default constructor
12     Array(const Array&);     // copy constructor
13     ~Array();               // destructor
14     int getSize() const;    // return size
15     const Array &operator=(const Array&); // assignment operator
16     bool operator==(const Array &)const; // equality operator
17     bool operator!=(const Array &right) const{ // inequality operator
18         return !(*this==right); // invokes Array::operator==
19     }
20     // subscript operator for non-const objects returns modifiable lvalue
21     int &operator[](int);
22     // subscript operator for const objects return rvalue
23     int operator[](int) const;
24 private:
25     int size;               // pointer-based array size
26     int *ptr;               // pointer to the first element
27 };
28 #endif
```

```
1 #include <iostream>
2 #include <iomanip>
3 #include <cstdlib>
4 using namespace std;
5 #include "Array.h"
6
7 Array::Array(int arraySize){
8     size = (arraySize > 0 ? arraySize : 10);
9     ptr = new int[size];
10    for (int i=0; i<size; i++) ptr[i]= 0;
11 }
12
13 Array::Array(const Array &arrayToCopy):size(arrayToCopy.size){
14     ptr = new int[size];
15     for (int i=0; i<size; i++)
16         ptr[i] = arrayToCopy.ptr[i];
17 }
18
19 Array::~Array(){
20     delete[] ptr;
21 }
22 int Array::getSize() const{
23     return size;
24 }
25 bool Array::operator==(const Array &right) const {
26     if (size != right.size) return false;
27     for (int i=0; i<size; i++)
28         if (ptr[i] != right.ptr[i]) return false;
29     return true;
30 }
```

```
31
32 int &Array::operator[](int subscript){
33     if (subscript < 0 || subscript >= size){
34         cerr << "\nError: Subscript " << subscript << " out of range" << endl;
35         exit(1);
36     }
37 }
38
39 int Array::operator[](int subscript) const{
40     if (subscript < 0 || subscript >= size){
41         cerr << "\nError: Subscript " << subscript << " out of range" << endl;
42         exit(1);
43     }
44     return ptr[subscript];
45 }
46
47 istream &operator>>(istream&input, Array&a){
48     for (int i=0; i<a.size; i++) input >> a.ptr[i];
49     return input;
50 }
51
52 ostream &operator<<(ostream &output, const Array &a){
53     int i;
54     for (i=0; i<a.size; i++){
55         output << setw(12) << a.ptr[i];
56         if ((i+1)%4==0) output << endl;
57     }
58     if (i%4 !=0) output << endl;
59     return output;
60 }
```

```
1 #include <iostream>
2 #include "Array.h"
3 using namespace std;
4
5 int main(){
6     Array integers1(7);
7     Array integers2;
8     cout << "Size of Array integers 1 is " << integers1.getSize()
9     << "\nArray after initialization\n" << integers1;
10    cout << "Size of Array integers 2 is " << integers2.getSize()
11    << "\nArray after initialization\n" << integers2;
12
13    cout << "\nEnter 17 integers: " << endl;
14    cin >> integers1 >> integers2;
15
16    cout << "\nAfter input, the Arrays contain: \n"
17    << "integers1:\n" << integers1 << "integers2:\n" << integers2;
18
19    cout << "\nEvaluating: integers != integers2" << endl;
20    if (integers1 != integers2)
21        cout << "integers1 and integers2 are not equal" << endl;
22
23    Array integers3(integers1);
24    cout << "\nSize of Array integers3 is " << integers3.getSize()
25    << "\nArray after initialization \n" << integers3;
26    return 0;
27 }
```



Size of Array integers 1 is 7

Array after initialization

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | |

Size of Array integers 2 is 10

Array after initialization

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | | |

Enter 17 integers:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the Arrays contain:

integers1:

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | |

integers2:

| | | | |
|----|----|----|----|
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | | |

Evaluating: integers != integers2

integers1 and integers2 are not equal

Size of Array integers3 is 7

Array after initialization

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | |

LECTURE 1

Software Engineering Observation



Software Engineering Observation

The argument to a copy constructor should be a **const** reference to allow a **const** object to be copied.



Common Programming Error

- Note that a copy constructor *must* receive its argument by reference, not by value.
- Otherwise, the copy constructor call results in infinite recursion (a fatal logic error) because receiving an object by value requires the copy constructor to make a copy of the argument object.



Common Programming Error

- If the copy constructor simply copied the pointer in the source object to the target object's pointer, then both objects would point to the same dynamically allocated memory.
- The first destructor to execute would then delete the dynamically allocated memory, and the other object's `ptr` would be undefined, a situation called a **dangling pointer**—this would likely result in a serious run-time error (such as early program termination) when the pointer was used.



Error-Prevention Tip

- Returning a reference from an overloaded `<<` or `>>` operator function is typically successful because `cout`, `cin` and most stream objects are global, or at least long-lived.
- Returning a reference to an automatic variable or other temporary object is dangerous—creating “dangling references” to nonexisting objects.

LECTURE 1

Operator ++ and --

Overloading ++ and --

- Increment/decrement operators can be overloaded
 - Suppose we want to add 1 to a Date object, d1
 - Prototype (member function)
 - `Date &operator++();`
 - `++d1` becomes `d1.operator++()`
 - Prototype (global function)
 - `Date &operator++(Date &);`
 - `++d1` becomes `operator++(d1)`



Overloading ++ and --

- To distinguish prefix and postfix increment
 - Postfix increment has a dummy parameter
 - An `int` with value `0`
 - Prototype (member function)
 - `Date operator++(int);`
 - `d1++` becomes `d1.operator++(0)`
 - Prototype (global function)
 - `Date operator++(Date &, int);`
 - `d1++` becomes `operator++(d1, 0)`



Overloading ++ and --

- Return values
 - Prefix increment
 - Returns by reference (Data &)
 - *lvalue* (can be assigned)
 - Postfix increment
 - Returns by value
 - Returns temporary object with old value
 - *rvalue* (cannot be on left side of assignment)
- All this applies to decrement operators as well

LECTURE 1

Case Study: Date Class



Case Study: A `Date` Class

- Example `Date` class
 - Overloaded increment operator
 - Change day, month and year
 - Overloaded `+=` operator
 - Function to test for leap years
 - Function to determine if day is last of month

```

1  #ifndef DATE_H
2  #define DATE_H
3  #include <iostream>
4  using namespace std;
5  class Date{
6      friend ostream &operator<<(ostream&, const Date&);
7  public:
8      Date(int m=1, int d=1, int y=1900); // default constructor
9      void setDate(int, int, int); // set month, day, year
10     Date &operator++(); // prefix increment operator
11     Date operator++(int); // prefix increment operator
12     const Date &operator+=(int); // add days, modify object
13     bool leapYear(int) const; // is date in a leap year
14     bool endOfMonth(int) const; // is date at the end of a month
15 private:
16     int month;
17     int day;
18     int year;
19     static const int days[]; // array of days per month
20     void helpIncrement(); // utility function for incrementing date
21 };
22 #endif

```

```

2 #include "Date.h"
3 const int Date::days[] = {
4     0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
5 };
6 Date::Date(int m, int d, int y){ setDate(m, d, y); }
7 void Date::setDate(int mm, int dd, int yy){
8     month = (mm>=1 && mm<=12) ? mm : 1;
9     year = (yy>=1900 && yy<=2100) ? yy : 1900;
10    if (month== 2 && leapYear(year)) day=(dd>=1 && dd<=29)? dd:1;
11    else day=(dd>=1 && dd<=days[month])? dd:1;
12 }
13 Date &Date::operator++(){
14     helpIncrement();
15     return *this;
16 }
17 Date Date::operator++(int){
18     Date temp = *this;
19     helpIncrement();
20     return temp;
21 }
22 const Date &Date::operator+=(int additionalDays){
23     for (int i=0; i<additionalDays; i++) helpIncrement();
24     return *this;
25 }
26
27 bool Date::leapYear(int testYear) const{
28     if (testYear%400==0 || (testYear!=0&&testYear%4==0))
29         return true;
30     else
31         return false;
32 }

```

```

27 bool Date::leapYear(int testYear) const{
28     if (testYear%400==0 || (testYear!=0&&testYear%4==0))
29         return true;
30     else
31         return false;
32 }
33 bool Date::endOfMonth(int testDay) const{
34     if (month==2&& leapYear(year))
35         return testDay==29;
36     else return testDay==days[month];
37 }
38 void Date::helpIncrement(){
39     if (!endOfMonth(day)) day++;
40     else if (month<12){
41         month++; day=1;
42     }
43     else{
44         year++; month=1; day=1;
45     }
46 }
47 ostream &operator<<(ostream &output, const Date &d){
48     static char *monthName[13] = {
49         "", "January", "February", "March", "April",
50         "May", "June", "July", "August", "September",
51         "October", "November", "December"
52     };
53     output << monthName[d.month] << ' ' << d.day << ", " << d.year;
54     return output;
55 }

```

```

1 #include <iostream>
2 using namespace std;
3 #include "Date.h"
4
5 int main(int argc, char** argv) {
6     Date d1;
7     Date d2(12, 27, 1992);
8     Date d3(0, 99, 8045);
9     cout<< "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is " << d3;
10    cout << "\n\nd2 += 7 is " << (d2+=7);
11
12    d3.setDate(2, 28, 1992);
13    cout << "\n\n d3 is " << d3;
14    cout << "\n++d3 is " << ++d3 << "(leap year allows 29th)";
15
16    Date d4(7, 13, 2002);
17    cout << "\n\nTesting the prefix increment operator: \n"
18    << " d4 is " << d4 << endl;
19    cout << "++d4 is " << ++d4 << endl;
20    cout << " d4 is " << d4;
21    cout << "\n\nTesting the postfix increment operator:\n"
22    << " d4 is " << d4 << endl;
23    cout << "d4++ is " << d4++ << endl;
24    cout << " d4 is " << d4 << endl;
25    return 0;
26 }

```

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993
```

```
    d3 is February 28, 1992
++d3 is February 29, 1992(leap year allows 29th)
```

Testing the prefix increment operator:

```
    d4 is July 13, 2002
++d4 is July 14, 2002
    d4 is July 14, 2002
```

Testing the postfix increment operator:

```
    d4 is July 14, 2002
d4++ is July 14, 2002
    d4 is July 15, 2002
```