C++

Operator Overloading

CO658 Data Structures & Algorithms

Topics

- Operator Overloading
  - Unary
  - Binary
- Overloading & member functions
- Vectors & Algorithms
- Recursion

Overload Operators

- C++ supports operators for native types (+,-,*,/,= etc)
- The operators are overloaded.
- Their behaviour is different depending upon the arguments.
- Operators are treated as polymorphic functions (operator functions)
- Not all operators can be overloaded
- Unary, Binary and Conversion operators can be overloaded
  - Unary operators have a single operand (++, -- etc)
  - Binary have two operands (+ , - , < , > etc)
  - Conversion – Casting
- New operators can’t be created

Unary

- An operator with a single operand
- The syntax for an operator @

```
type operator@(const type& identifier) {   }
```

- The statements in the {} will implement the rules of the operator and return a variable of type.

```
identifier1  = @identifier2
```

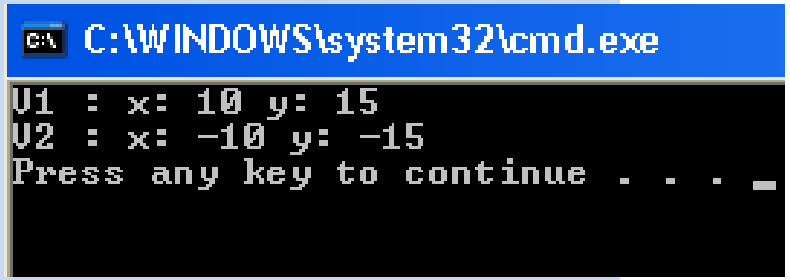
- When invoking the operator identifier2 becomes the argument to the formal parameter identifier.
- Usually we don’t want to change this value so it is const.
- Identifier1 is assigned the return value of the overloaded operator.
- The following example demonstrates how the ! operator is overloaded for a 2D vector containing an x & y coordinate. To return a vector in the opposite direction.

Unary Operator Overload Example

```
class Vector2D {
public:
        int x,y;
        Vector2D(){}
        Vector2D(int px, int py):x(px),y(py){}
};

Vector2D operator!(const Vector2D& vec){
        Vector2D result;
        result.x = (vec.x * -1);
        result.y = (vec.y * -1);
        return result;
}

int main(){
    Vector2D v1(10,15);
    Vector2D v2;
    cout << "V1 : x: " << v1.x << " y: " << v1.y << endl;
    v2 = !v1;
    cout << "V2 : x: " << v2.x << " y: " << v2.y << endl;
}
```



Unary Example

- While enumerate types are assigned an integer value (0..n-1), we can not use the ++ operator to increment the value.

```
enum Day { sun, mon, tue, wed, thu, fri, sat };

int main(){
        Day myDay = tue;
        myDay++;
        cout << myDay << endl;
}
```

- The above code will NOT run and generates a ++ not defined error.
- We need to overload the ++ to define what should happen when applied to a variable of the Day type.

Unary Example

- The new function overloads the operator ++
- The single operand myDay is passed as a reference to the parameter d.
- d is modified and since it is a reference, myDay is updated with the new value.

```
enum Day { sun, mon, tue, wed, thu, fri, sat };

Day operator++(Day& d){
        return d = (sat==d) ? sun : Day(d+1);
}

int main(){
        Day myDay = tue;
        ++myDay;
        cout << myDay << endl;  // Displays 3
}
```

- The return value has no bearing on the operation.

Unary Example

- The increment and decrement operators are special cases as the same operator can be applied postfix or prefix.

```
myDay++ ;   // Postfix
++myDay;    // Prefix
```

- To distinguish the overloaded operators a dummy int parameter is inserted in the postfix overload version.

```
enum Day { sun, mon, tue, wed, thu, fri, sat };

Day operator++(Day& d, int){
        return d = (sat==d) ? sun : Day(d+1);
}

int main(){
        Day myDay = tue;
        myDay++;
        cout << myDay << endl;
}
```

Binary

- Have two arguments
- Arguments do not have to be the same type

```
type operator@(const type& left, const type& right) { }
```

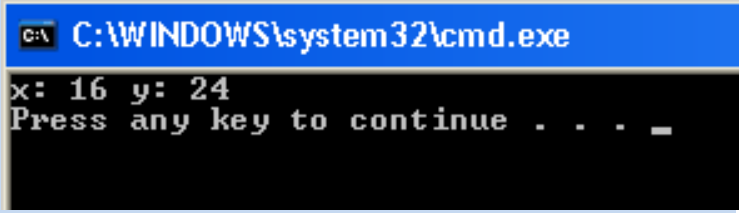
- The following example demonstrates how the + operator is overloaded for a 2D vector containing an x & y coordinate.

Example

```
class Vector2D {
public:
        int x,y;
        Vector2D(){}
        Vector2D(int px, int py):x(px),y(py){}
};

Vector2D operator+(const Vector2D& left, const Vector2D& right){
        Vector2D result;
        result.x = left.x + right.x;
        result.y = left.y + right.y;
        return result;
}

int main(){
        Vector2D v1(10,15);
        Vector2D v2(6,9);
        Vector2D v3 = v1 + v2;
        cout << "x: " << v3.x << " y: " << v3.y << endl;
}
```



Binary

- The comparison operator could also be overloaded.
- This operator returns a Boolean value

```
bool operator==(const Vector2D& left, const Vector2D right){
        return (left.x == right.x && left.y == right.y);
}
```

Constant operand

- If the operand/s should not be changed during the execution of the operation, then add the const keyword.

```
bool operator==(const Vector2D& left, const Vector2D right){
        return (left.x == right.x && left.y == right.y);
}
```

- Trying to change the operands within the overload function will cause a compiler error.

Overloading Inside a Class

- Operators can be overloaded as member functions.
- When declared as a member function the parameters are reduced by one, as the calling object is implicitly supplied.
- This refers to the object on the left of the operator.
- Binary operators take one explicit parameter and unary none.

Unary member function

```
type ClassName::operator @(){ }
```

Binary member function

```
type ClassName::operator @(const ClassName& right){ }
```

Vector2D.h

```
class Vector2D {
public:
        int x,y;
        Vector2D(){}
        Vector2D(int px, int py):x(px),y(py){}
        Vector2D operator!();
        Vector2D operator+(const Vector2D& );
};

Vector2D Vector2D::operator!(){
        Vector2D result;
        result.x = (this->x * -1);
        result.y = (this->y * -1);
        return result;
}

Vector2D Vector2D::operator+(const Vector2D& vec){
        Vector2D result;
        result.x = this->x + vec.x;
        result.y = this->y + vec.y;
        return result;
}
```

Overloading Inside a Class

- The first operand must be of the class type.
- An operator can’t exclusively operate on pointers
- So p1 < p2 where p1 and p2 are pointers could not be overloaded
- The pointer/s could however be dereference

```
*ptr1 < *ptr2
```

- C++ returns a reference to the object when the pointer is dereferenced.

Activity

Attempt exercise 1:

- Create an **Entity** class
- Overload the following operators so they can compare the size of Entity objects (instead of integers):
  - ++(prefix)
  - --(prefix)
  - >
  - <

Vector

- Class that encapsulates the behaviour of an array.
- C++ alternative to C arrays
- Elements are stored in contiguous memory locations.
- Elements accessed by either index position or offset pointer.
- The **capacity** (allocated storage space) of the vector is managed automatically so it can grow and shrink as demand requires.
- The **size** (number of elements) in the vector is always such that the memory requirement is less than the capacity.
- Capacity is increased in blocks so that a single insertion would not necessarily incur a reallocation.
- Have a similar performance to **arrays ????**

Vector

- Defined within C++ standard library

```
#include <vector>

using namespace std;
```

- A template class

```
vector<type> identifier;
```

- Optionally the vector can be constructed with a specified number of elements, which will be initialised to their default value.

```
vector<type> identifier(size);
```

- If the elements are to be initialised to a non default value, this can be specified in the constructor.

```
vector<type> idenifier(size,defaultValue);
```

Vector

- The elements within a vector can be accessed using the subscript operator [ ]

```
identifier[index]
```

- We can iterate through the vector as if it were an array

```
const int SIZE = 10;
vector<int> data(SIZE);
for(int n=0;n < SIZE; n++){
        data[n] = n;
}
```

push_back

- As an alternative to using the subscript operator the vector's push_back() member function can be invoked to add a new value to one past the last element.

```
idenifier.push_back(value)
```

- Note: if a reallocation occurs this will invalidate any existing iterators.

```
vector<int> data2;
for(int n=0;n < 15; n++)
        data2.push_back(n);
}
```

Iterator

- Mechanism for traversal of data structures.
- Uses a pair of pointers (iterator pair) that identify the beginning and end of the data to be traversed.
- The vector class contains **begin** and **end** member functions that return a pointer to the first and one past the last element respectively.

```
vector<type>::iterator identifier;
```

- Within a for loop the pointer can be incremented and dereferenced to obtain the value it points to

```
vector<int>::iterator it;

for (it = data.begin();it != data.end(); it++)
        cout << *it << endl;
```

Algorithms

- A generic set of template algorithms
- Implemented as stand alone functions

```
#include <algorithm>

using namespace std;
```

- Functions include find, copy, sort, fill, min, max

find

- Returns an iterator to the first matching value within the data structure.
- Note: The returned iterator will enable you to traverse the vectors list from the first matching value until the end.
- Returns an iterator to the last + 1 element if not found.

```
vector<T>::iterator  identifier;
identifier = find(vectorIdentifier.begin, vectorIdentifier.end, value);
```

- To search the whole vector begin and end would be the vector's corresponding pointers.

```
vector<int>::iterator fit;
fit = find(data.begin(),data.end(), 6);
```

- Where data is a vector.

Find & Overloading

- User defined types will require an overloaded equality operator if you intend to use the find function.
- Find will not match object references when determining if two objects are the same.
- The iterator is a pointer to the object so the object's data members and functions can be accessed through it.

```
it->member
```

erase

- Vector member function that removes a single element or range of elements from a vector.
- The vector will automatically reposition the elements to maintain a contiguous set of elements. This will involve making copies of the objects.
- Returns the element after the one that's deleted.
- Invalidates any iterators.

```
vectorIdentifier.erase(deleteIteratorIdentifier);
```

- Usually used in conjunction with find to locate the element to be erased.

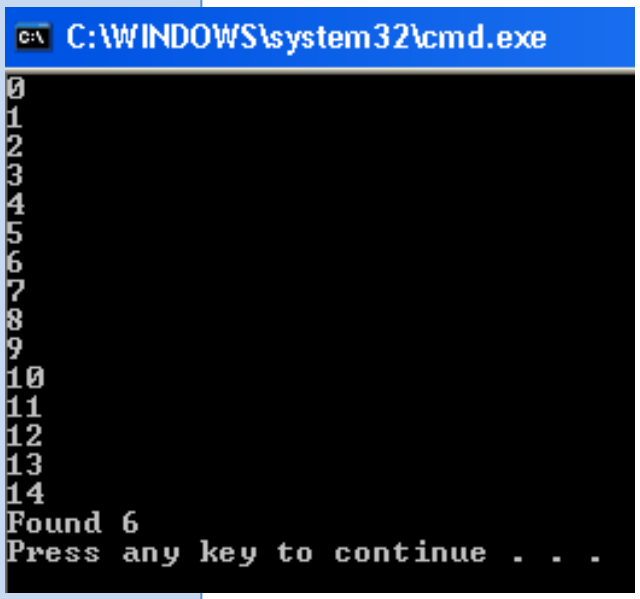
```
vector<int>::iterator fit;
fit = find(data.begin(),data.end(), 6);
vector<int>::iterator eit;
eit = data.erase(fit);
```

Vector Example

```
#include "stdafx.h"
#include <vector>
#include <iostream>


using namespace std;

int main(){
		vector<int> data;
		for(int n=0;n < 15; n++){
		    data.push_back(n);
		}
		vector<int>::iterator it;
		for (it = data.begin();it != data.end(); it++)
		      cout << *it << endl;
		vector<int>::iterator fit;
		fit = find(data.begin(),data.end(), 6);
		if (fit != data.end())
		     cout << "Found " << *fit << endl;
		return 0;
}
```



Activity

Attempt exercise 2 and 3:

- Define a **Weapon** class
- Create 10 weapon objects
- Assign IDs for each of the weapons using an iterator
- Use an iterator to print the values to screen
- Then find the weapon with the ID of 5, and delete it afterwards

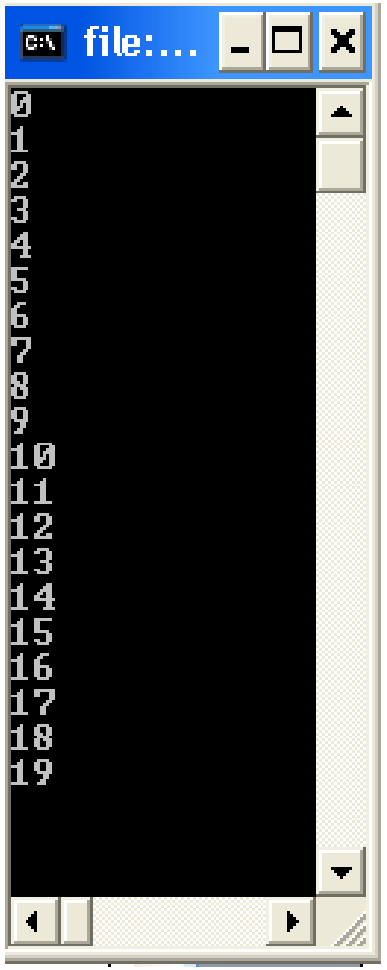
Recursion

- A technique used extensively to navigate over data structures.
- A recursive function is one that calls itself.
- The function must contain logic to ensure it stops the recursive calls.
- If not it will generate a stack overflow exception.

Recursion Example

```
class Numbers {
public:
    void DisplayLessThan(int n) {
        --n;
        if (n > 0) {
            DisplayLessThan(n);
        }
        cout << n << endl;
    }
};

int main(){
            Numbers num;
            num.DisplayLessThan(20);
            return 0;
}
```



Activity

Attempt exercise 4 and 5:

- Define a **IOObject** class
- Create two methods which calculate the factorial of a number
  - One method which uses iteration
  - One method which uses recursion

Summary

- Operator overloading provides an intuitive mechanism for manipulating objects.
- It use should improve the readability of the code.
- Vector is a flexible C++ class that provides an alternative to C style arrays