Member Operators

Eden Burton <ronald.burton@senecacollege.ca>
github repository:
(https://github.com/Seneca-OOP244/SCD-Notes)

Assignment Operator

"..defines logic for copying data from an existing object to another existing object .."

- syntax,
  `ClassName& operator= (const ClassName&)`
- called when assignment statement is executed
  `harry = joe;`
- differs from copy constructor because both objects currently exist
- definition design
  - checks for self-assignment (i.e. `harry = harry`)
  - shallow copies the non-resource instance variables
  - deallocates current object resources
  - allocate a new resource for the current object
  - copies resource data associated with the source object to the newly allocated resource memory of the current object

Localization

- code for object initialization is virtually the same
  - copy constructor: `Student harry(joe)`
  - assignment operator: `harry = joe`
- avoid code duplication by
  - define private method
  - call method in both copy constructor and assignment operator method

Localization Example

```
Student::Student(const Student& source) {
   grade = nullptr; *this = source;
}

 Student& Student::operator=(const Student& src) {
   if (this != &source) {
   no = src.no;
   numOfGrades = src.numOfGrades
   delete [] grade;
   if (src.grade != nullptr) {
      grade = new int[numOfGrades];
      for (int i = 0; i < numOfGrades; i++)
                grade[i] = source.grade[i];
   } else grade = nullptr;
   return *this;
 }}
```

Operator Overloading

- often we have an assumed notation of an operator
- usually based on fundamental types

```
int main ()  {
   int i,j;
   i = 2; j = 3;
   cout << "are␣i␣and␣j␣the␣same:␣"
   <<   (i == j) << endl;

   i = 2; j = 3;
   Student harry, joe;

   cout << "are␣harry␣and␣joe␣the␣same:␣"
   <<   (harry == joe) << endl;

   cout << "␣harry␣+␣joe␣=␣␣"
   <<   (harry + joe) << endl;
}
```

Operator Overloading

- C++ allows refining the meaning of operators for compound types

```
struct Point {
    int x,y;
    bool operator==(const Point&);
    Point& operator+(const Point&);
};

bool Point::operator==(const Point& rSide) {
    return ((x==rSide.x) && (y==rSide.y));
}

Point& Point::operator+(const Point& rSide) {
    x += rSide.x; y += rSide.y; return *this;
}
```

Operator Overloading

```
#include <iostream>
#include "Point.h"
using namespace std;

int main() {
    Point p1={2,3}; Point p2={4,5};
    Point p3 = p1 + p2;
    cout << "Equal: " << (p1 == p2) << endl;
    cout << "x: " << p3.x
         << " y: " << p3.y << endl;
}
```

Free Helpers

- does not need access to private members
- all information accessed through the public interface

```
class Point {
    int x,y;
    bool operator==(const Point&);
    Point& operator+(const Point&);
    int getX() const {return x;};
    int getY() const {return y;};
};

areSame(const Point& lSide, const Point& rSide) {
    return  (lSide.getX() == rSide.getX())
            && (lSide.getY() == rSide.getY()) ;
}
```

Friends

- grants access to private members
- violate encapsulation rules

```
class Point {
    int x,y;
    friend bool operator==(const Point& lhs,
                           const Point& lhs );
    Point& operator+(const Point&);
};

bool Point::operator==(const Point& lSide,
                       const Point& rSide) {
    return ((lSide.x==rSide.x)
            && (lSide.y==rSide.y));
}
```