CS 162
Intro to Programming II
Operator Overloading
Operator Overloading

• One operator used for different operations or actions
• Why we need it?
• How it’s used?
Automatic Type Conversion

• When you overload operators, there is a subtle automatic type conversion you need to be aware of

• We will illustrate this with a point class, which represents a 2 dimensional point
Automatic Type Conversion

point.hpp

#ifndef POINT_HPP
#define POINT_HPP

class point {
public:
  point(int x_value, int y_value) :
    x(x_value),
    y(y_value) {
  }
  int get_x() const { return x; }
  int get_y() const { return y; }

private:
  int x;
  int y;
};

const point operator +(const point& p1, const point& p2);
#endif
Automatic Type Conversion

point.cpp

#include <iostream>
#include "point.hpp"

const point operator +(const point& p1, const point& p2) {
    return point(p1.get_x() + p2.get_x(),
                 p1.get_y() + p2.get_y());
}

int main(int argc, char** argv) {
    point p1(1,2);
    point p2(3,4);
    point p3 = p1+p2;
    std::cout << p3.get_x() << " " << p3.get_y() << std::endl;
}
Automatic Type Conversion

• Now change the main function to look like the following:

```cpp
int main(int argc, char** argv) {
    point p1(1,2);
    point p2(3,4);
    point p3 = p1+10;
    std::cout << p3.get_x() << " " << p3.get_y() << std::endl;
}
```

The code won’t compile. Why? The operator + only accepts objects of type point and 10 is an int.
Automatic Type Conversion

- Add a constructor to point.hpp that takes a single int as an argument

```cpp
class point {
public:
    point(int x_value) : x(x_value), y(0) {}
    point(int x_value, int y_value) : x(x_value), y(y_value) {}
    int get_x() const { return x; }
    int get_y() const { return y; }
private:
    int x;
    int y;
};
const point operator +(const point& p1, const point& p2);
```
Automatic Type Conversion

• If you now try to compile this main function, it will compile just fine!

```cpp
int main(int argc, char** argv) {
    point p1(1,2);
    point p2(3,4);
    point p3 = p1+10;
    std::cout << p3.get_x() << " " << p3.get_y() << std::endl;
}
```

• It will output: 11 2
Automatic Type Conversion

• C++ will automatically convert 10 into an object of type point
• C++ will call the constructor for point that takes a single int as an argument
• This creates a point with x = 10 and y = 0
Operator Overloading

• Non-member operator overloading

/* In point.cpp */
const point operator +(const point& p1, const point& p2) {
    return point(p1.get_x() + p2.get_x(), p1.get_y() + p2.get_y());
}

/* In point.hpp */
class point {
    /* etc. */
};

const point operator +(const point& p1, const point& p2);
Operator Overloading

• Member operator overloading

/* In point.cpp */
const point point::operator +(const point& p2) {
    return point(x + p2.get_x(), y + p2.get_y());
}

/* In point.hpp */
class point {
public:
    /* etc. */
    const point operator +(const point& p2);
    /* etc. */
};
Operator Overloading

• Non-member operator overloading

Pros:
Can interchange the order of the arguments to the operator eg.

- p1+10 or
- 10+p1

Cons:
Can’t access member variables directly and incurs overhead of a getter function call
Operator Overloading

• Member operator overloading

Pros:
• Can access member variables directly

Cons:
• Can’t interchange the order of the arguments to the operator
  – p1+10 is allowed because p1 is the calling object and 10 is the argument
  – 10 + p1 is not allowed because 10 is not of type Point
Operator Overloading

• Can we get the best of both worlds?
• Yes, using overloading as a friend function
Friends

• **Friends** of a class have access to private member variables and member functions of that class

• Two types of friends:
  – Friend functions
  – Friend classes
Friends

• Overloading as a friend function

/* In point.hpp */
class point {
public:
    point(int x_value) : x(x_value), y(0) {
    }
    point(int x_value, int y_value) : x(x_value), y(y_value) {
    }
    int get_x() const { return x; }
    int get_y() const { return y; }
    friend const point operator +(const point& p1, const point& p2);
private:
    int x;
    int y;
};
Friends

• Overloading as a friend function

/* In point.cpp */
const point operator +(const point& p1, const point& p2)
{
    return point(p1.x + p2.get_x(), p1.y + p2.get_y());
}

Note: the operator is outside the class and has access to member variables x and y
Friends

• You can also declare a class A to be a friend of class B
• This means A has access to the member variables and member functions of B
class A;

class B
{
 public:
  // etc.
  friend class A;
};

class A
{
  // etc.
};
Advanced Operator Overloading

• You can also overload operators such as:
  
  `<<`
  
  `>>`
  
  `=
  
  `[ ]`

• But be warned...
Overloading `<<`

- The insertion operator `<<` is typically used as follows:
  ```cpp
  std::cout << "Hello world" << std::endl;
  ```
- If we rewrite this in our operator "function" notation we get:
  ```cpp
  <<(std::cout, "Hello world")
  ```
Overloading `<<`

- What if we replace the second parameter with a point object `p1`
  
  `<<(std::cout, p1)`

- This will cause a compile error unless we overload the `<<` operator

- This makes it easy to print out point objects!
Overloading `<<`

/* In point.hpp */
class point {
public:
    /* etc. */
    friend std::ostream& operator <<(std::ostream& output_stream, const point& p);

private:
    /* etc. */
};
Overloading `<<`

/* In point.cpp */
std::ostream& operator <<(std::ostream& output_stream,
    const point& p) {
    std::cout << "x = " << p.x << " , y = " << p.y << 
    std::endl;
    return output_stream;
}

int main(int argc, char** argv) {
    point p1(1,2);
    point p2(3,4);
    point p3 = p1+p2;
    std::cout << p3 << std::endl;
}

Prints- x = 4 , y = 6
Overloading `<<`

• Note: The overloaded operator `<<` returns a reference of type ostream. Why?

• Allows you to chain `<<` statements eg.
  `std::cout << p3 << "and more stuff " << std::endl;`

• Think of the above like the following:
  `(((std::cout << p3) << "and more stuff ") << std::endl)`

• All 40 operators that can be overloaded
Operator Overloading

• One operator used for different operations or actions
• Why we need it?
• How it’s used?
• New access- friend