Applying Our Knowledge of Classes

Introducing C++ Friends
Overloading Additional Operators

Review of Inheritance

Member access specifiers

- public – accessed by anybody
- private – accessed by only Base functions or friends
  - This means derived classes can't access base class private members directly
- protected – allows Derived classes to access members, but not accessible outside the class

Base access specifiers

- Can also be public, private or protected
- Be sure to review the assigned reading

```
class Base {
public:
    int pub;
private:
    int pri;
protected:
    int pro;
};
```

```
class Derived : public Base {
public:
    Derived(){
        pub = 1; //allowed
        pri = 2; //not allowed
        pro = 3; //allowed
    }
};
```

```
int main(){
    Base b;
    b.pub = 1; //allowed
    b.pri = 2; //not allowed
    b.pro = 3; //not allowed
    Derived d;
    d.pub = 1; //allowed
    d.pri = 2; //not allowed
    d.pro = 3; //not allowed

}
```

Meeting new friends (in C++)

- Functions or classes declared with the **friend** keyword
  - Allows non-member function to access the protected and private members of a class
- Can also have entire friend classes
  - In this case, a class can share private variables with another class even if there is no inheritance
- Not a two way street! Both classes must be friends if you want to share private variables in both directions.

Example of friendship syntax

- Can be normal function, or member function of another class

Syntax

- "friend" in front of function prototype
- Can be in private, public, or protected

```
class Point {
private:
    int x;
    int y;
public:
    Point();
    Point(int, int);
    ~Point();
    Point(const Point&);
    void move_left(int);
    friend void print_location(const Point&);
};

void print_location(const Point &point){
    cout << "Location is " << point.x << "," << point.y
    <<endl
}

int main() {
    Point p1(1,2);
    Point p2 = p1;
    print_location(p2);
}   return 0;
```

Friend Classes & Member Functions

- Gives all members of friend class access to private members of the other class
- Friend class has no direct access to "this" pointer of other class objects
- Remember: If you want two classes to be friends of each other, both must declare the other as a friend

```
class Point {
private:
    int x;
    int y;
public:
    Point();
    Point(int, int);
    ~Point();  Point(const
    Point&);  void
    move_left(int);
    friend class Line;
    friend void Line::createLine(const Point&, const Point&);
};
```

Why use friendship?

- At first glance, this seems like a bad idea.
  - Violates the principle of encapsulation
  - Don’t be tempted to use friends to avoid writing good code

- Can be useful in some cases
  - See example code of overloading the “<<” operator

Overloading operators

- We have already seen the assignment operator overload
- In C++, nearly every operator can be overloaded
  - This can simplify your life and make it easier to interact with objects
- Sample operators (not a complete list):
  - =, ++, -=, <<, >>, +, /, ->
- See the assigned reading for examples
  - https://en.cppreference.com/w/cpp/language/operators

Overloading - Details

Vocab Refresh:
Precedence determines which operators are evaluated first
Associativity determines the direction in which operators are evaluated (left or right)

- Can only overload operators that exist
- At least one of the operands in an overloaded operator must be a user-defined type
- Can't change the number of operands an operator supports
- Can't change the default precedence and associativity (C++ order of operations)
    - https://en.cppreference.com/w/cpp/language/operator_precedence

Best Practices:

- Keep the function of the operators close to the original intent as possible
- If the meaning isn't clear and intuitive, use a named function instead

Vocab Refresh:
Unary operators act on single operands to make new value
Binary operators act on two operands to make new value

How to Overload...

- 3 ways to overload operators
  - Member function – mostly used when modifying left operand
  - Friend function – convenient due to direct access to class members
  - Normal function – considered to be best, but only works if you have accessors/getters
- How to choose?
  - If **=**, **[]**, **()**, or **->**, must use member function
  - If unary operator (e.g. ++), use member function
  - If binary operator that doesn’t modify left operand (e.g. +), use normal/friend
  - If binary operator that does modify left operand, but can’t change definition of it, use normal/friend (<<)
  - If binary operator that does modify left operand, but you can change definition of it, use member function (+=)

Operator Overloading Application – I/O

- Printing each member variable of a class on the screen can be annoying

```
cout << “Point” << p.get_x() << “,”<< p.get_y() << endl;
```

- Have used print member functions to get around this

```
void print() {
    cout << “Point” << x << “,” << y << endl;
}
p.print();
```

- Overload the << operator!

```
friend ostream& operator<<(ostream& out, const Point& p) {
    out << “Point” << p.x << “,” << p.y << endl;
    return out;
}

cout << p << endl;
```