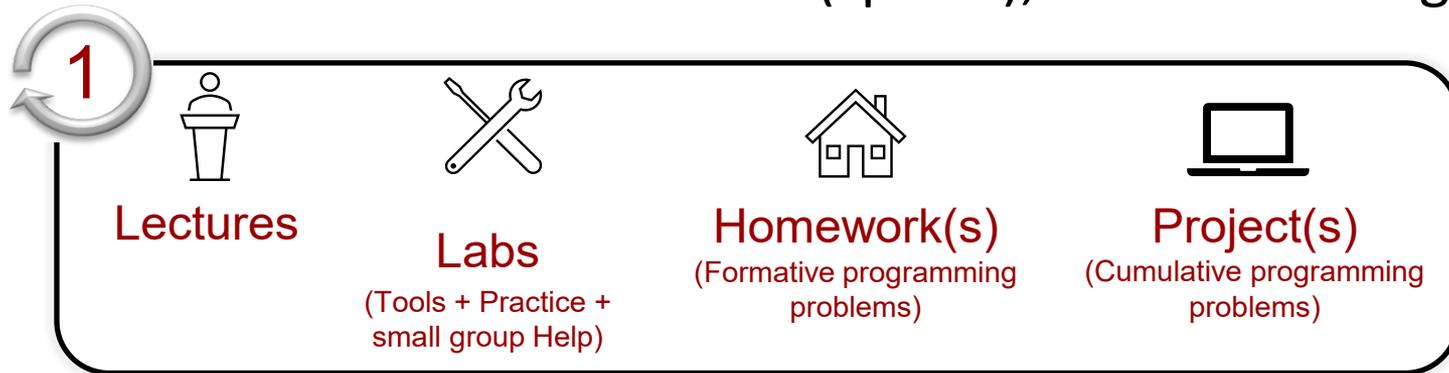


CS 103 Unit 5a – Operator Overloading

Unit 5 – Objects Part 2

- The course is broken into 6 units (spirals), each consisting of:



**C++ Language
Syntax**



**Algorithms and
Computational Thinking**



Objects 1



Managing Data



Objects 2



Recursion

Function Overloading

- What makes up a signature (uniqueness) of a function
 - name
 - number and type of arguments
- No two functions are allowed to have the same signature; the following 4 functions are unique and allowable...
 - `void f1(int);` `void f1(double, int);`
 - `void f1(double);` `void f1(int, int);`
- We say that “f1” is **overloaded** 4 times
- Notes:
- Return type does NOT make signature unique
 - `int f1();` is considered the same as `void f1();`
- For member functions, 'const' make signature unique
 - `int& List::get()` `int const & List::get() const;`

Operator Overloading

- C/C++ defines operators (+,*,-,,=,etc.) that work with basic data types like `int`, `char`, `double`, etc.
- C/C++ has no clue what classes we'll define and what those operators would mean for these "yet-to-be-defined" classes

- class complex {
 public:
 double real, imaginary;
 };

- Complex c1,c2,c3;

 // should add component-wise
 c3 = c1 + c2;

- class List {
 ...
 };

- List l1,l2;

 l1 = l1 + l2; // should concatenate
 // l2 items to l1

- **We can write custom functions to tell the compiler what to do when we use these operators! Let us learn how...**

```
class User{
public:
    User(string n); // Constructor
    string get_name();
private:
    int id_;
    string name_;
};
```

user.h

```
#include "user.h"
User::User(string n) {
    name_ = n;
}
string User::get_name(){
    return name_;
}
```

user.cpp

```
#include<iostream>
#include "user.h"

int main(int argc, char *argv[]) {
    User u1("Bill"), u2("Jane");
    // see if same username
    // Option 1:
    if(u1 == u2) cout << "Same";

    // Option 2:
    if(u1.get_name() == u2.get_name())
        { cout << "Same" << endl; }
    return 0;
}
```

user_test.cpp

Two Approaches

- There are two ways to specify an operator overload function
 - 1. Global level function (not a member of any class)
 - 2. As a member function of the class on which it will operate
- Which should we choose?
 - It depends on the **left-hand side** operand
 - Ex 1: **string** + int
 - Ex 2: **iostream** << Location

Method 1: Global Functions

- Can define global functions with name "`operator{+-...}`" taking two arguments
 - LHS = Left Hand side is 1st arg
 - RTH = Right Hand side is 2nd arg
- When compiler encounters an operator with objects of specific types it will look for an "operator" function to match and call it
- But what if we need to access private data of some object to implement our operation?
 - A global (non-member) function will not work. We need method 2

```
int main()
{
    int hour = 9;
    string suffix = "p.m.";

    string time = hour + suffix;
    // WON'T COMPILE...doesn't know how to
    // add an int and a string
    return 0;
}
```

```
string operator+(int time, string suf)
{
    string res = to_string(time); // conv int to str
    res += suf; // strings already support +
    return res;
}
int main()
{
    int hour = 9;
    string suffix = "p.m.";

    string time = hour + suffix;
    // WILL COMPILE TO:
    // string time = operator+(hour, suffix);

    return 0;
}
```

Method 2: Class Members

- C++ allows users to write class member functions that define what an operator should do for a class
- Same naming convention: function name starts with 'operator' and then the actual operator
- Important: **Left-hand side** is the implied calling object for which the member function is called and **Right-hand side** is passed as the argument
 - **LHS-arg.operator+(RHS-arg);**

```
class Complex
{ public:
  Complex();
  Complex(double r, double i);
  Complex operator+(const Complex &rhs) const;

 private:
  double real, imag;
};

Complex Complex::operator+(const Complex &rhs) const
{
  Complex temp;
  temp.real = real + rhs.real;
  temp.imag = imag + rhs.imag;
  return temp;
}

int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = c1 + c2;
  // Same as c3 = c1.operator+(c2);
  cout << c3.real << "," << c3.imag << endl;
  // can overload '<<' so we can write:
  // cout << c3 << endl;
  return 0;
}
```

Overloading Notes

- You can overload any operator except the member operator (`.`), the scope operator (`::`), and the ternary operator (`? :`)
 - **Binary operators:** `+`, `-`, `*`, `/`, `++`, `--`
 - **Comparison operators:** `==`, `!=`, `<`, `>`, `<=`, `>=`
 - **Assignment:** `=`, `+=`, `-=`, `*=`, `/=`, etc.
 - **I/O stream operators:** `<<`, `>>`
- You cannot change the operator's precedence
 - Multiply must always come before addition
- More questions: <https://isocpp.org/wiki/faq/operator-overloading>

Binary Operator Overloading

- For binary operators, do the operation on a new object's data members and return that object
 - Don't want to affect the input operands data members
 - Difference between: $x = y + z;$ vs. $x = x + z;$
- Normal order of operations and associativity apply (can't be changed)
- Can overload each operator with various RHS types...
 - See next slide

Binary Operator Overloading

```
class Complex
{
public:
    Complex();
    Complex(double r, double i);
    Complex operator+(const Complex &rhs) const;
    Complex operator+(int real) const;
private:
    double real, imag;
};

Complex Complex::operator+(const Complex &rhs) const
{
    Complex temp;
    temp.real = real + rhs.real;
    temp.imag = imag + rhs.imag;
    return temp;
}

Complex Complex::operator+( int real ) const
{
    Complex temp = *this;
    temp.real += real;
    return temp;
}
```

No special code is needed to add 3 or more operands. The compiler chains multiple calls to the binary operator in sequence.

```
int main()
{
    Complex c1(2,3), c2(4,5), c3(6,7);

    Complex c4 = c1 + c2 + c3;
    // (c1 + c2) + c3
    // c4 = c1.operator+(c2).operator+(c3)
    //      = anonymous-ret-val.operator+(c3)

    c3 = c1 + c2;
    c3 = c3 + 5;
}
```

Adding different types
(Complex + Complex vs.
Complex + int) requires
different overloads

Relational Operator Overloading

- Can overload
==, !=, <, <=, >, >=
- Should return **bool**

```
class Complex
{
public:
    Complex();
    Complex(double r, double i);
    Complex operator+(const Complex &rhs) const;
    bool operator==(const Complex &rhs) const;
    double real, imag;
};

bool Complex::operator==(const Complex &rhs) const
{
    return (real == rhs.real && imag == rhs.imag);
}

int main()
{
    Complex c1(2,3);
    Complex c2(4,5);
    // equiv. to c1.operator==(c2);
    if(c1 == c2)
        cout << "C1 & C2 are equal!" << endl;

    return 0;
}
```

Nothing will be displayed

Non-Member Functions

- What if the user changes the order?
 - int on LHS & Complex on RHS
 - No match to a member function b/c to call a member function the LHS has to be an instance of that class
- We can define a non-member function (global scope function) that takes in two parameters (both the LHS & RHS)
 - May need to declare it as a friend

```
int main()
{
    Complex c1(2,3);
    Complex c2(4,5);
    Complex c3 = 5 + c1;
                // ?? 5.operator+(c1) ??
                // ?? int.operator+(c1) ??
                // there is no int class we can
                // change or write

    return 0;
}
```

Doesn't work without a new operator+ overload

```
Complex operator+(const int& lhs, const Complex &rhs)
{
    Complex temp;
    temp.real = lhs + rhs.real;    temp.imag = rhs.imag;
    return temp;
}
int main()
{
    Complex c1(2,3);
    Complex c2(4,5);
    Complex c3 = 5 + c1;    // Calls operator+(5,c1)
    return 0;
}
```

Still a problem with this code

Can operator+(...) access Complex's private data?

Friend Functions

- A friend function is a function that is not a member of the class but **has access to the private data members of instances** of that class
- Put keyword **'friend'** in function prototype in class definition
- Don't add scope to function definition

```
class Silly
{
public:
    Silly(int d) { dat = d };
    friend int inc_my_data(Silly &s);
private:
    int dat;
};

// don't put Silly:: in front of inc_my_data(...)
// since it isn't a member of Silly
int inc_my_data(Silly &a)
{
    s.dat++;
    return s.dat;
}

int main()
{
    Silly cat(5);
    //cat.dat = 8
    // WON'T COMPILE since dat is private

    int x = inc_my_data(cat);
    cout << x << endl;
}
```

Notice `inc_my_data` is NOT a member function of `Silly`. It's a global scope function but it now can access the private class members.

Non-Member Functions

- Revisiting the previous problem

```
class Complex
{
public:
    Complex();
    Complex(double r, double i);
    // this is not a member function
    friend Complex operator+(const int&, const Complex& );
private:
    double real, imag;
};

Complex operator+(const int& lhs, const Complex &rhs)
{
    Complex temp;
    temp.real = lhs + rhs.real;    temp.imag = rhs.imag;
    return temp;
}

int main()
{
    Complex c1(2,3);
    Complex c2(4,5);
    Complex c3 = 5 + c1;    // Calls operator+(5,c1)
    return 0;
}
```

Now things work!

Why Friend Functions?

- Can I do the following?
- error: no match for 'operator<<' in 'std::cout << c1'
- /usr/include/c++/4.4/ostream:108: note: candidates are: /usr/include/c++/4.4/ostream:165: note: std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT, _Traits>::operator<<(long int) [with _CharT = char, _Traits = std::char_traits<char>]
- /usr/include/c++/4.4/ostream:169: note: std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT, _Traits>::operator<<(long unsigned int) [with _CharT = char, _Traits = std::char_traits<char>]
- /usr/include/c++/4.4/ostream:173: note: std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT, _Traits>::operator<<(bool) [with _CharT = char, _Traits = std::char_traits<char>]
- /usr/include/c++/4.4/bits/ostream.tcc:91: note: std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT, _Traits>::operator<<(short int) [with _CharT = char, _Traits = std::char_traits<char>]

```
class Complex
{
public:
    Complex();
    Complex(double r, double i);
    Complex operator+(const Complex &rhs) const;
private:
    double real, imag;
};

int main()
{
    Complex c1(2,3);
    cout << c1; // equiv. to cout.operator<<(c1);
    cout << endl;
    return 0;
}
```

Why Friend Functions?

- `cout` is an object of type 'ostream'
- `<<` is just an operator
- But we call it with `cout` on the LHS which would make `operator<<` a member function of class `ostream`
- `Ostream` class can't define these member functions to print out user defined classes because they haven't been created
- Similarly, `ostream` class doesn't have access to private members of `Complex`

```
class Complex
{
public:
    Complex();
    Complex(double r, double i);
    Complex operator+(const Complex &rhs) const;
private:
    double real, imag;
};

int main()
{
    Complex c1(2,3);
    cout << "c1 = " << c1;
    // cout.operator<<("c1 = ").operator<<(c1);

    // ostream::operator<<(const char *str);
    // ostream::operator<<(Complex &src);

    // Using global scope (friend) functions
    // operator<<(operator<<(cout,"c1 = "), c1);

    cout << endl;
    // operator<<(cout,endl);
    return 0;
}
```

Ostream Overloading

- Can define operator functions as friend functions
- LHS is 1st arg.
- RHS is 2nd arg.
- Use friend function so LHS can be different type but still access private data
- Return the ostream& (i.e. os which is really cout) so you can chain calls to 'operator<<' and because cout/os object has changed

```
class Complex
{
public:
    Complex();
    Complex(double r, double i);
    Complex operator+(const Complex &rhs) const;
    friend ostream& operator<<(ostream&, const Complex &c);
private:
    int real, imag;
};

ostream& operator<<(ostream &os, const Complex &c)
{
    os << c.real << "," << c.imag << "j";
    //cout.operator<<(c.real).operator<<(",").operator<<...
    return os;
}

int main()
{
    Complex c1(2,3), c2(4,5);
    cout << c1 << c2;
    // operator<<( operator<<(cout, c1), c2);
    cout << endl;
    return 0;
}
```

Implicit Type Conversion

- Would the following if condition make sense?
- No! If statements want Boolean variables
- But you've done things like this before
 - Operator >> returns an ifstream&
- So how does ifstream do it?
 - With an "implicit type conversion operator overload"
 - Student::operator bool()
 - Code to specify how to convert a Student to a bool
 - Student::operator int()
 - Code to specify how to convert a Student to an int

```
class Student {
    private: int id; double gpa;
};
int main()
{
    Student s1;
    if(s1){ cout << "Hi" << endl; }
    return 0;
}
```

```
ifstream ifile(filename);
...
while( ifile >> x )
{ ... }
```

```
class Student {
    private:
        int id; double gpa;
    public:
        operator bool() { return gpa >= 2.0; }
        operator int() { return id; }
};

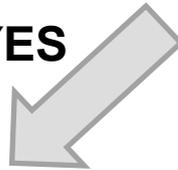
Student s1;
if(s1) // calls operator bool() and
    int x = s1; // calls operator int()
```

Member or Friend?

Should I make my operator overload be a member of a class, C1?

Ask yourself: *Is the LHS an instance of C1?*

YES



```
C1 objA;  
objA << objB // or  
objA + int
```

YES the operator overload function can be a **member function** of the C1 class since it will be translate to `objA.operator<<(...)`

NO



```
C1 objA;  
objB << objA // or  
int + objA
```

NO the operator overload function should be a **global level (maybe friend) function** such as `operator<<(cout, objA)`. It cannot be a member function since it will be translate to `objB.operator<<(...)`.

Summary

- If the **left hand side** of the operator is an **instance of that class**
 - Make the operator a **member function of a class...**
 - The member function should only take in one argument which is the RHS object
- If the **left hand side** of the operator is an **instance of a different class**
 - Make the operator a **friend function of a class...**
 - This function requires two arguments, first is the LHS object and second is the RHS object

OPERATOR OVERLOADING REVIEW

Operator Overloading Review

Return types

- For class BigInt which models an arbitrary precision integer, what should the return type be for:
 - Operator+
 - Operator==

```
class BigInt {
public:
    _____ operator+(const BigInt&);
    _____ operator==(const BigInt&);
};
int main(){
    BigInt w, x, y, z;
    w = x + y;
}
```

Chaining

- Do we need operator overload functions with 2-, 3-, 4-inputs, etc. to handle various use cases?

```
class BigInt {
    ...
};
int main(){
    BigInt w, x, y, z;
    w = x + y + z;
    cout << w << " is a bigint!" << endl;
}
```

SOLUTION

