CS-202

C++ Classes – Operator(s) (Pt.2)

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday		Sunday
			Lab (8 Sections)			
	CLASS		CLASS			
PASS Session	PASS Session	Project DEADLINE	NEW Project	PASS Session		PASS Session

Your 4th Project Deadline is next Wednesday 2/27.

- PASS Sessions held Friday-Sunday-&-Monday-Tuesday, get all the help you need!

Today's Topics

C++ Classes Cheatsheet

- Declaration
- Members, Methods, Interface
- Implementation – Resolution Operator (`::`)
- Instantiation – Objects
- Object Usage – Dot Operator (`.`)
- Object Pointer Usage – Arrow Operator (`->`)
- Classes as Function Parameters, Pass-by-Value, by-(`const`)-Reference, by-Address
- Protection Mechanisms – `const` Method signature
- Classes – Code File Structure
- Constructor(s), Initialization List(s), Destructor
- `static` Members – Variables / Functions
- Operator Overloading

Class `friend`(s)

Keyword `this`

Operator Overloading (continued)

Class Cheatsheet

Operator Overloading – **non-Member** of Class.

- **Unary** Operator(s):

```
const Money operator-(const Money & mn)
{ return Money(-mn.getD(),-mn.getC()); }
Money myMoney(99,25), notMyMoney = - myMoney;
```

- **Binary** Operator(s):

```
bool operator==(const Money & mn1, const Money & mn2)
{ return mn1.getD()==mn2.getD() && mn1.getC() == mn2.getC(); }

const Money operator+(const Money& mn1, const Money& mn2)
{ return Money(mn1.getD()+mn2.getD(),mn1.getC()+mn2.getC()); }
Money myMoney(99,25), yourMoney(0,75);
bool ourMoneyEqual = myMoney == yourMoney;
Money ourMoney = myMoney + yourMoney;
```

return: a **const** *Unnamed* Class Object

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(const Money & m);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars;
  int m_cents;
};
```

Note:
Operator(s) should handle Class specifications (e.g. prevent ***m_cents*** rollover)

Class Cheatsheet

Operator Overloading – **Class Member** Function.

➢ **Assignment** Operator (half the story, the rest for later) :

```
void Money::operator=(const Money & mn)
{ m_dollars = mn.m_dollars; m_cents = mn.m_cents; }
Money myMoney(99,25), myMoneyAgain = myMoney;
```

A Class method, like saying: `myMoneyAgain.operator=(myMoney);`

Note: If none specified, compiler creates a default Assignment Operator (***Member***-*Copy*) for Class Objects. *Remember*: ***Shallow***-Copy vs ***Deep***-Copy.

➢ **Binary** Operator(s):

```
const Money Money::operator+(const Money & mn) const
{ return Money(m_dollars+mn.m_dollars, m_cents+mn.m_cents); }
Money myMoney(99,25), yourMoney(0,75);
Money ourMoney = myMoney + yourMoney;
```

Calling Object is like 1st parameter: `myMoney.operator+(yourMoney);`

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(const Money & m);
  void Money operator=
 (const Money & m);
  const Money operator+
 (const Money & m) const;
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars;
  int m_cents;
  char * m_owner;
};
```

Class Cheatsheet

➢ Operator Overloading – Both versions (*Ambiguous*):

```
const Money operator+(const Money &a,const Money &b)
{  return Money(1);  }  //Non-class Method
const Money Money::operator+(const Money & b)  const
{  return Money(2);  } //Class Method
warning: ISO C++ says that these are ambiguous …
Money m1,m2, m3 = m1 + m2;
Money m4 = m1 .operator+ ( m2 );
```

Result: 1

Result: 2

➢ Operator Overloading – Both versions (*Different Calls*):

```
const Money operator-(const Money & mn)
{ return Money(-mn.getD(), -mn.getC()); }
const Money Money::operator-(const Money & m)  const
{ return Money(m_dollars-mn.m_dollars, m_cents-mn.m_cents); }
Money m5 = - m1 ;       //Unary call  - Non-class Method
Money m6 = m1 - m2 ;    //Binary call - Class Method
```

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(const Money & m);
const Money operator+
(const Money & m)  const;
const Money operator-
(const Money & m)  const;
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars;
  int m_cents;
};
```

Class Cheatsheet

Operator Overloading

- Return by-**const**-Value

```
const Money Money::operator+(const Money & mn)const{
  return Money(m_dollars + mn.m_dollars,
               m_cents   + mn.m_cents);
}
```

Why **const**-Value ?

```
Money a(4, 50), b(3, 25), c(2, 10);
```

`(a + b);`	Evaluates to: *Unnamed* Object
`c = (a + b);`	OK…
`(a + b) = c;`	No !!!

Prevents (&protects) us from altering the returned value…

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(const Money & m);
  void operator=
 (const Money & m);
  const Money operator+
 (const Money & m) const;
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars;
  int m_cents;
};
```

Class Cheatsheet

Operator Overloading

- Return by-**const**-Reference (?)

```
const Money& Money::operator+(const Money& mn) const
{ return Money(m_dollars + mn.m_dollars,
               m_cents   + mn.m_cents);  }
```

```
warning: returning reference to temporary.
```

- Makes a temporary Object, which goes out of scope!

```
Money a(4, 50), b(3, 25);
const Money * ab_Pt = &(a + b);
```

```
cout << ab_Pt->getD()
<<","<< ab_Pt->getC();
```

```
7
75
```

No !
This is UNSAFE !

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(const Money & m);
  void Money operator=
 (const Money & m);
  const Money & operator+
 (const Money & m) const;
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars, m_cents;
};
```

Function **return** does not guarantee an immediate *Stack* frame wipe!

Note: Especially if the return type is *not* a **const**-Reference ! (…)

Class Cheatsheet

Operator Overloading

- Return by-Reference – Operator ([])

Returned: <type_id> &, internal Member Reference.

```
int & Money::operator[](unsigned int index)
{ return m_transID [ index ]; }
```

- Accessing (private) Data Member by-Reference:

```
Money hugeCheck(1000000);
unsigned int transCnt = 0;
hugeCheck [ transCnt++ ] = BANK_TRANS;
hugeCheck [ transCnt++ ] = BRIBE_TRANS;
hugeCheck [ transCnt++ ] = BANK_TRANS;
if (hugeCheck [ 1 ] == BRIBE_TRANS)
{ cout << "Illegal Activity!"; }
```

Write-to

Read-from

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(const Money & m);
  int & operator[](
      unsigned int index);
  const Money& operator+
 (const Money & m) const;
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars, m_cents;
  int m_transID[T_HIST];
};
```

Friend Functions

Remember: Operator Overloads as no-Member function:

- Access of data through *Accessor* and *Mutator* functions.
- Very inefficient (call overhead).

Class **`friend`**(s) can directly access **`private`** Class data.

- Any function can be a Class **`friend`**.
- Make non-Member Operator Overloads **`friend`**(s) (no overhead, more efficient).

Operator Overloads as non-Member Class **`friend`**(s).

- Most common use (avoids need to go through Setter / Getter functions interface).
- Need data access anyway.

Class Friend(s)

Friend Functions

A **`friend`** Function of a Class is:

- *Not* a Member Function, but still has direct access to **`private`** members.
- Specified in Class Declaration (keyword **`friend`**) but still isn't a Member Function.

Friends and *Purity*:

- "Spirit" of OOP dictates all Operators and Functions must be Member Functions. (many believe **`friend`**(s) violate basic OOP principles.

However: Very advantageous for Operators:

- Allow automatic type conversion.
- Encapsulation is retained – **`friend`** is in Class Declaration.
- Improves efficiency.

Friend Classes

A **friend** Class of another Class:

- Has direct access to **private** members.
- Is specified in Class Declaration (keyword **friend**).

Example: **class** ***F*** is **friend** of **class** ***C***

- All **class** ***F*** Member Functions are **friend**s of **class** ***C***.
- Not reciprocated relationship, **friend**ship granted, not taken!

```
class C {
  friend bool operator==(const C& c1,const C& c2); //A friend regular function
  friend class F;  //A friend class (all its member functions are friends)
  …
}
```

Functions have direct access to any private Members
(Data and Functions of **class** ***C***)

Return by-Reference – Cascading

Remember: Overloading Operator (`[]`)

Get `<type_id>&`, internal Member Reference:

```
int & Money::operator[](unsigned int index) {
  return m_transID[index];
}
```

```
class Money{
 int & operator[](unsigned int i);
 int m_transID[T_HIST];
};
```

Another utility for Operator Overloading:

➢ Cascading (daisy-chaining):

```
double & chainableFun(double & var) {
  var += 1.0;
  return var;
}
```

```
double x;
chainableFun(chainableFun(…(chainableFun(x))…));
```

Note: Cannot do `return var+1.0;`

`double&+double` has no Reference!

(it's a **rvalue** – and that's our limit – for now …)

```
error: invalid initialization of non-
const reference of type 'double&' from
an rvalue of type 'double'
```

Return by-Reference – Cascading

Overloading Operator (**<<**):

- Insertion (*Binary*) Operator.
- Used with ***cout*** Object (from **<iostream>** library).

Example:

```
cout << "Hello world!";
```

2nd Operand: C-string Literal

1st Operand: **class** ***ostream*** Object

Operator Overloading

Return by-Reference – Cascading

Overloading Operator (**<<**):

- Insertion (*Binary*) Operator.
- Used with ***cout*** Object (from **<iostream>** library).

Example:

```
cout << "Hello world!";
```

2nd Operand: C-string Literal

1st Operand: **class** ***ostream*** Object

Instead of:

```
hugeCheck.output();
```

We can overload it for a **class** ***Money*** type 2nd operand:

```
Money hugeCheck(1000000,0);
cout << hugeCheck;
```

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  friend ostream &
  operator<<(ostream & os,
             const Money & m);
  void output();
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars, m_cents;
};
```

Operator Overloading

Return by-Reference – Cascading

Overloading Operator (**<<**):

- Insertion (*Binary*) Operator.
- Used with ***cout*** Object (from **<iostream>** library).

Cascading how-to: Return by-1st-operand-Reference.

```
ostream& operator<<(ostream& os, const Money& mn){
   os << "$" << mn.m_dollars << "." << mn.m_cents;
   return os;
}
```

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  friend ostream &
  operator<<(ostream & os,
          const Money & m);
  void output();
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars, m_cents;
};
```

Operator Overloading

Return by-Reference – Cascading

Overloading Operator (**<<**):

- Insertion (*Binary*) Operator.
- Used with ***cout*** Object (from **<iostream>** library).

Cascading how-to: Return by-1st-operand-Reference.

```
ostream& operator<<(ostream& os, const Money& mn){
   os << "$" << mn.m_dollars << "." << mn.m_cents;
   return os;
}

Money myMoney(99,25), yourMoney(0,75);
cout <<"Mine:"<<myMoney<<" Yours:"<<yourMoney;
```

Like calling:

```
operator<<(operator<<(operator<<(operator<<(cout
,"Mine:"),myMoney)," Yours:"),yourMoney);
```

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  friend ostream &
  operator<<(ostream & os,
             const Money & m);
  void output();
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars, m_cents;
};
```

Operator Overloading

Return by-Reference – Cascading

Overloading Operator (**>>**):

- Extraction (*Binary*) Operator.
- Used with ***cin*** Object (from **<iostream>** library).

Overloading and Cascading (return by-1st-operand-Ref).

```
istream& operator>>(istream& is, Money& mn){
   char dollarChar;
   is >> dollarChar;
   if (dollarChar=='$'){
      double dollarsDouble;
      is >> dollarsDouble;
      mn.m_dollars = dollarsDouble;
      mn.m_cents = 100*(dollarsDouble-mn.m_dollars);
   }
   return is;
}
```

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  friend ostream &
  operator<<(ostream & os,
             const Money& m);
  friend istream &
  operator>>(istream & is,
                 Money& m);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars,m_cents;
};
```

Operator Overloading

Overloading Operators (<<) , (>>)

```
1   #include <iostream>
2   #include <cstdlib>
3   #include <cmath>
4   using namespace std;

5   //Class for amounts of money in U.S. currency
6   class Money
7   {
8   public:
9       Money( );
10      Money(double amount);
11      Money(int theDollars, int theCents);
12      Money(int theDollars);
13      double getAmount( ) const;
14      int getDollars( ) const;
15      int getCents( ) const;
16      friend const Money operator +(const Money& amount1, const Money& amount2)
17      friend const Money operator -(const Money& amount1, const Money& amount2)
18      friend bool operator ==(const Money& amount1, const Money& amount2);
19      friend const Money operator -(const Money& amount);
20      friend ostream& operator <<(ostream& outputStream, const Money& amount);
21      friend istream& operator >>(istream& inputStream, Money& amount);
22  private:
23      int dollars; //A negative amount is represented as negative dollars and
24      int cents; //negative cents. Negative $4.50 is represented as -4 and -50.
25      int dollarsPart(double amount) const;
26      int centsPart(double amount) const;
27      int round(double number) const;
28  };
```

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  friend ostream &
  operator<<(ostream & os,
          const Money& m);
  friend istream &
  operator>>(istream & is,
                Money& m);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars,m_cents;
};
```

Operator Overloading

Overloading Operators (<<) , (>>)

```
ostream& operator <<(ostream& outputStream, const Money& amount)
{
    int absDollars = abs(amount.dollars);
    int absCents = abs(amount.cents);
    if (amount.dollars < 0 || amount.cents < 0)
        //accounts for dollars == 0 or cents == 0
        outputStream << "$-";
    else
        outputStream << '$';
    outputStream << absDollars;
    if (absCents >= 10)
        outputStream << '.' << absCents;
    else
        outputStream << '.' << '0' << absCents;

    return outputStream;
}

//Uses iostream and cstdlib:
istream& operator >>(istream& inputStream, Money& amount)
{
    char dollarSign;
    inputStream >> dollarSign; //hopefully
    if (dollarSign != '$')
    {
        cout << "No dollar sign in Money input.\n";
        exit(1);
    }

    double amountAsDouble;
    inputStream >> amountAsDouble;
    amount.dollars = amount.dollarsPart(amountAsDouble);
    amount.cents = amount.centsPart(amountAsDouble);

    return inputStream;
}
```

In the main function, cout is plugged in for outputStream.

For an alternate input algorithm, see Self-Test Exercise 3 in Chapter 7.

Returns a reference

In the main function, cin is plugged in for inputStream.

Since this is not a member operator, you need to specify a calling object for member functions of Money.

Returns a reference

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  friend ostream &
  operator<<(ostream & os,
             const Money& m);
  friend istream &
  operator>>(istream & is,
                 Money& m);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars,m_cents;
};
```

Operator Overloading

Overloading Operators (<<) , (>>)

```
29  int main( )
30  {
31      Money yourAmount, myAmount(10, 9);
32      cout << "Enter an amount of money: ";
33      cin >> yourAmount;
34      cout << "Your amount is " << yourAmount << endl;
35      cout << "My amount is " << myAmount << endl;
36
37      if (yourAmount == myAmount)
38          cout << "We have the same amounts.\n";
39      else
40          cout << "One of us is richer.\n";

41      Money ourAmount = yourAmount + myAmount;
42      cout << yourAmount << " + " << myAmount
43           << " equals " << ourAmount << endl;

44      Money diffAmount = yourAmount - myAmount;
45      cout << yourAmount << " - " << myAmount
46           << " equals " << diffAmount << endl;

47      return 0;
48  }
```

Since << returns a reference, you can chain << like this. You can chain >> in a similar way.

Sample Dialogue

```
Enter an amount of money: $123.45
Your amount is $123.45
My amount is $10.09.
One of us is richer.
$123.45 + $10.09 equals $133.54
$123.45 - $10.09 equals $113.36
```

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  friend ostream &
  operator<<(ostream & os,
             const Money& m);
  friend istream &
  operator>>(istream & is,
                 Money& m);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars,m_cents;
};
```

Operator Overloading

Operators (++) , (--) (half the story, the rest for later)

Overloading Pre-Increment Operator(s):

- No arguments (for compiler *disambiguation*).

```
Money Money::operator++(){
  m_cents++;
  if (m_cents…){ m_dollars=…; m_cents=…; } //and fix
  return Money(m_dollars, m_cents);
}
```

```
class Money{
public:
 Money();
 Money(int d, int c=0);
 Money(const Money &m);
 Money operator++();
 Money operator--();
 Money operator++(int);
 Money operator--(int);
 void setD/C(int dc);
 int getD/C() const;
private:
 int m_dollars,m_cents;
};
```

Operator Overloading

Operators (++) , (--) (half the story, the rest for later)

Overloading Pre-Increment Operator(s):

- No arguments (for compiler *disambiguation*).

```
Money Money::operator++(){
  m_cents++;
  if (m_cents…){ m_dollars=…; m_cents=…; } //and fix
  return Money(m_dollars, m_cents);
}
```

Note:
Modifies calling Object and **return**s a Copy of it.

```
Money myMoney(0,99);
Money myMoreMoney = ++ myMoney;
```

{1,0} {1,0}

```
class Money{
public:
 Money();
 Money(int d, int c=0);
 Money(const Money &m);
 Money operator++();
 Money operator--();
 Money operator++(int);
 Money operator--(int);
 void setD/C(int dc);
 int getD/C() const;
private:
 int m_dollars,m_cents;
};
```

Operator Overloading

Operators (**++**) , (**--**) (half the story, the rest for later)

Overloading Post-Increment Operator(s):

- A dummy **int** argument (for compiler *disambiguation*).

```
Money Money::operator++(int dummy){
  Money moneyCopy(m_dollars, m_cents);
  m_cents++;
  if (m_cents…){ m_dollars=…; m_cents=…; } //fix
  return moneyCopy;
}
```

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  Money operator++();
  Money operator--();
  Money operator++(int);
  Money operator--(int);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars, m_cents;
};
```

Operator Overloading

Operators (++) , (--) (half the story, the rest for later)

Overloading Post-Increment Operator(s):

- A dummy `int` argument (for compiler *disambiguation*).

```
Money Money::operator++(int dummy){
   Money moneyCopy(m_dollars, m_cents);
   m_cents++;
   if (m_cents…){ m_dollars=…; m_cents=…; } //fix
   return moneyCopy;
}
```

Note: Keeps a Copy of calling Object to `return` and then modifies calling Object.

```
Money myMoney(0,99);
Money mySameMoney = myMoney ++;
```

{0,99} {1,0}

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  Money operator++();
  Money operator--();
  Money operator++(int);
  Money operator--(int);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars, m_cents;
};
```

Keyword this

A Pointer to the Calling Object.

- Inside a Class Member Function, we can address the Calling Object itself (and its members) "by-name" !
- Keyword this provides a way to address the entire Calling Object inside a Member Function call.

```
Money & Money::thisFunction(){
  this -> m_dollars = 1000; //access data
  this -> setC(99);         //call a method
  return *this;
}
```

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  Money & thisFunction();
  Money operator++();
  Money operator--();
  Money operator++(int);
  Money operator--(int);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars, m_cents;
};
```

Keyword this

A Pointer to the Calling Object.

- Inside a Class Member Function, we can address the Calling Object itself (and its members) "by-name" !
- Keyword this provides a way to address the entire Calling Object inside a Member Function call.

```
Money & Money::thisFunction(){
   this -> m_dollars = 1000; //access data
   this -> setC(99);         //call a method
   return *this;
}
```

Note: A Member Function can return a Reference to the Calling Object that invoked it.

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  Money & thisFunction();
  Money operator++();
  Money operator--();
  Money operator++(int);
  Money operator--(int);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars, m_cents;
};
```

Keyword this

Overloading Pre-Increment Operator(s) (now for the rest):

- No arguments (for compiler *disambiguation*).

```
Money & Money::operator++(){
  m_cents++; … //mutates calling object
  return *this;
}
```

Note:
Modifies calling Object and **return**s a Reference to it.
No Object Copy operation!

```
Money myMoney(0,99);
Money myMoreMoney = ++ myMoney;
```

{100,0} {100,0}

```
class Money{
public:
 Money();
 Money(int d, int c=0);
 Money(const Money &m);
 Money & operator++();
 Money & operator--();
 Money operator++(int);
 Money operator--(int);
 void setD/C(int dc);
 int getD/C() const;
private:
 int m_dollars, m_cents;
};
```

Keyword this

Overloading Post-Increment Operator(s) (now for the rest) :

- A dummy `int` argument (for compiler *disambiguation*).

```
Money Money::operator++(int dummy){
  Money moneyCopy(*this);
  this->m_cents++; … //mutates calling object
  return moneyCopy;
}
```

Note: Keeps a Copy of calling Object to **return** and then modifies calling Object (same as before).

```
Money myMoney(0,99);
Money mySameMoney = myMoney ++;
```

{99,0} {100,0}

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  Money & operator++();
  Money & operator--();
  Money operator++(int);
  Money operator--(int);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars, m_cents;
};
```

Keyword this

Checking if the Calling Object is *exactly* the same as the Object passed as argument!

```
bool Money::thisCheck(const Money & m){
  if (this == &m)
    return true;
  else
    return false;
}
```

➢ Example: To protect from (unwillingly) tampering with own self:

```
Money cashiers[100];
Money* active_desk = cashiers;
active_desk += rand_desk_offset;
```

```
for (int i=0; i<100; ++i)
 active_desk->accum(cashiers[i]);
```

Sums contents of all onto itself, but should avoid adding its own data twice (skip if Calling Object `active_desk` is the same as `cashiers[i]`)

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  bool thisCheck(const
                 Money & m);
  void accum(const
                 Money & m);
  Money & operator++();
  Money & operator--();
  Money operator++(int);
  Money operator--(int);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars, m_cents;
};
```

Keyword this

Overloading Assignment Operator (=) (now for the rest) :

- return Reference to Calling Object, maintain Assignment Operator sequencing:

```
Money & Money::operator=(const Money & rhs){
  m_dollars = rhs.m_dollars;
  m_cents = rhs.m_cents;
  return *this;
}
```

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  Money & operator=(const
                  Money & rhs);
  Money & operator++();
  Money & operator--();
  Money operator++(int);
  Money operator--(int);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars, m_cents;
};
```

Keyword this

Overloading Assignment Operator (=) (now for the rest) :

- return Reference to Calling Object, maintain Assignment Operator sequencing:

```
Money & Money::operator=(const Money & rhs){
  m_dollars = rhs.m_dollars;
  m_cents = rhs.m_cents;
  return *this;
}
```

- Maintain Assignment Operator sequencing:

```
Money a(4, 50), b(3, 25), c(2, 10);

a = b = c;
```

Can now do this too!

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  Money & operator=(const
                 Money & rhs);
  Money & operator++();
  Money & operator--();
  Money operator++(int);
  Money operator--(int);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars, m_cents;
};
```

Keyword this

Overloading Assignment Operator (=) (now for the rest) :

- Check if calling object is trying to assign from itself (right-hand-side (rhs) argument is the same Object) :

```
Money & Money::operator=(const Money & rhs){
  if (this != &rhs) {
    m_dollars = rhs.m_dollars;
    m_cents = rhs.m_cents;
  }
  return *this;
}
```

- Protect from Self-Assignment:

```
Money a(4, 50);
a = a;
```

- Avoid unnecessary assignments.
- Protect dynamically allocated data …

```
class Money{
 public:
  Money();
  Money(int d, int c=0);
  Money(const Money &m);
  Money & operator=(const
                 Money & rhs);
  Money & operator++();
  Money & operator--();
  Money operator++(int);
  Money operator--(int);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars, m_cents;
};
```

CS-202

Time for Questions !