The Header File

// Money1.h

// Money class declaration
#ifndef _MONEY1_H_
define _MONEY1_H_
class Money
{
public:
    Money(); // Default constructor
    Money(int d, int c = 0); // General constructor
    Money(double d); // Conversion constructor
    Money(const Money& m); // Copy constructor
    ~Money(); // Destructor

double total_value() const;

int get_dollars() const; // Accessors
int get_cents() const;

Money operator+(const Money& m) const; // Overloaded operators
Money operator-(const Money& m) const;
Money& operator++(); // Prefix
Money operator++(int); // Postfix
Money operator*(double d) const;
bool operator>(const Money& m) const;
bool operator>(double d) const;

operator int() const; // Conversion functions
operator double () const;

friend Money operator*(double d, const Money& m); // friend functions
friend istream& operator>>(istream& in, Money& m);
friend ostream& operator<<(ostream& out, const Money& m);
friend bool operator==(const Money& m1, const Money& m2);

private:
    int dollars;
    int cents;
};

// inline functions
inline int Money::get_dollars() const // Accessor
{
    return dollars;
}

inline int Money::get_cents() const // Accessor
{
    return cents;
}

inline double Money::total_value() const
{
    return dollars + 0.01 * cents;
}

#endif
The Implementation File

// Money1.cpp: Money1 Implementation File

#include <iostream> // cout
#include <cstdlib> // exit()
#include <iomanip> // setw(), setfill()

using namespace std;

#include "Money1.h"

Money::Money() // Default constructor
{
    dollars = cents = 0;
    cout << "DEFAULT CONSTRUCTOR CALLED\n";
}

Money::Money(int d, int c) // General constructor
{
    dollars = d;
    cents = c;
    cout << "GENERAL CONSTRUCTOR CALLED\n";
}

Money::Money(double d) // Conversion constructor
{
    dollars = d;
    cents = 100. * (d - dollars) + 0.5; // Round to nearest penny
    cout << "CONVERSION CONSTRUCTOR CALLED\n";
}

Money::Money(const Money& m) // Copy constructor
{
    dollars = m.dollars;
    cents = m.cents;
    cout << "COPY CONSTRUCTOR CALLED\n";
}

Money::~Money()
{
    cout << "DESTRUCTOR CALLED.\n";
}

Money Money::operator+(const Money& m) const // Pass by reference
{
    // Calls default constructor
    Money sum;
    sum.cents = (cents + m.cents) % 100;
    sum.dollars = dollars + m.dollars + (cents + m.cents) / 100;
    return sum; // Calls copy constructor;
    // Cannot return a reference; would be reference to local
    // object sum which goes out of scope at the return. It
    // would be a reference to a non-existent object.
}
Money Money::operator-(const Money& m) const
{
    Money diff; // Calls default constructor
    diff.dollars = dollars - m.dollars;
    diff.cents = cents - m.cents;

    if (diff.cents < 0)
    {
        --diff.dollars;
        diff.cents += 100;
    }

    if (diff.dollars < 0)
    {
        cerr << "Negative money value is not allowed.\n";
        exit(EXIT_FAILURE);
    }

    return diff; // Calls copy constructor
}

Money& Money::operator++() // Prefix: Increment, then Use
{
    ++dollars;
    return *this;
}

Money Money::operator++(int) // Postfix: Use, then Increment
{
    Money temp = *this; // Calls copy constructor
    dollars++;
    return temp; // Calls copy constructor
        // Cannot return local temp by reference
}

Money Money::operator*(double d) const
{
    double product = d * total_value();
    return Money(product); // Explicit call to conversion constructor;
        // return anonymous Money object that is created by constructor
        // Advantage: guaranteed that object is created according to standards
        // of constructor
}

bool Money::operator>(const Money& m) const
{
    return total_value() > m.total_value();
}

bool Money::operator>(double d) const
{
    return total_value() > d;
}
Money::operator int() const  // Noisy conversion functions
{
    cout << "operator int() CALLED\n";
    return dollars;
}

Money::operator double() const
{
    cout << "operator double() CALLED\n";
    return total_value();
}

// friend functions
Money operator*(double d, const Money& m)
{
    double product = d * m.total_value();  // or return m * d;
    return Money(product);  // Explicit call to conversion constructor
}

istream& operator>>(istream& in, Money& m)
{
    in >> m.dollars;
    in.ignore();  // ignore the decimal point
    in >> m.cents;
    return in;
}

ostream& operator<<(ostream& out, const Money& m)
{
    char old_fill = out.fill('0');  // returns old fill character
    out << '$'
        << m.dollars
        << '.'
        << setw(2)
        << m.cents
        << setfill(old_fill);
    return out;
}

bool operator==(const Money& m1, const Money& m2)
{
    if (m1.dollars == m2.dollars && m1.cents == m2.cents)
        return true;
    else
        return false;
}

bool operator==(const Money& m)
{
    return dollars == m.dollars &&
        cent == m.cents;
}

*/
The Client File

// Money1Client

#include <iostream>
using namespace std;
#include "Money1.h"

int main()
{
    Money m1, m2, m3; // calls default constructor
    Money m4(20, 25); // calls general constructor
    Money m5(50); // calls general constructor
    Money m6 = 30.96; // calls conversion constructor

    cout << "m1 is " << m1 << " m4 is " << m4
         << " m5 is " << m5 << " m6 is " << m6 << endl;

    m1 = m4 + m6; // m1 = m4.operator+(m6);
    m2 = m5 - m6; // m2 = m5.operator-(m6);
    m3 = ++m4; // m3 = m4.operator++();

    cout << "m1 is " << m1 << " m2 is " << m2
         << " m3 is " << m3 << " m4 is " << m4 << endl;

    // m1.operator++(1)
    cout << "m1++ is " << m1++ << " now m1 is " << m1 << endl;

    m1 = 65.0; // calls conversion constructor

    cout << "m3 * 2.5 is " << m3 * 2.5 << " 3.45 * m3 is " << 3.45 * m3
         << " m1 is " << m1 << endl;

    cout << boolalpha // m1.operator>(m5) m2.operator>(1.23)
         << "m1 > m5 is " << (m1 > m5) << " m2 > 1.23 is " << (m2 > 1.23)
         << " m4 == m6 is " << (m4 == m6) << endl; // (operator==((m4, m6))

    int iv = m2; // int iv = m2.operator int();
    double dv = m2; // double dv = m2.operator double();

    cout << "iv is " << iv << " dv is " << dv << endl;

    cout << "dollars of m2 is " << m2.get_dollars() << " cents of m2 is "
         << m2.get_cents() << " total value of m2 is "
         << m2.total_value() << endl;

    cout << "And now enter a Money type: ";
    cin >> m1; // operator>>(cin, m1);
    cout << "You entered " << m1 << endl;

    return 0;
}
Sample Run: (information in () was not generated by program)

277 DEFAULT CONSTRUCTOR CALLED (m1)
279 DEFAULT CONSTRUCTOR CALLED (m2)
280 DEFAULT CONSTRUCTOR CALLED (m3)
281 GENERAL CONSTRUCTOR CALLED (m4)
282 GENERAL CONSTRUCTOR CALLED (m5 with second argument defaulted to 0)
283 conversion constructor called (m6)
284 m1 is $0.00  m4 is $20.25  m5 is $50.00  m6 is $30.96
285 default constructor called (sum in operator+())
286 copy constructor called (temporary returned from operator+())
287 destructor called. (sum)
288 destructor called. (temporary returned from operator+())
289 default constructor called (diff in operator-())
290 copy constructor called (temporary returned from operator-())
291 destructor called. (diff)
292 destructor called. (temporary returned from operator-())
293 m1 is $51.21  m2 is $19.04  m3 is $21.25  m4 is $21.25
294 copy constructor called (initialize temp in operator++(int))
295 copy constructor called (temporary returned from operator++(int))
296 destructor called. (temp in operator++(int))
297 m1++ is $51.21 now m1 is $52.21
298 destructor called. (temporary returned from operator++(int))
299 conversion constructor called (m1 = Money(65.04);)
300 destructor called. (anonymous object from conv. constructor)
301 conversion constructor called (explicit call in operator*(double))
302 conversion constructor called (explicit call in operator*(double, Money))
303 m3 * 2.5 is $53.13  3.45 * m3 is $73.31  m1 is $65.00
304 destructor called. (temporary returned from operator*(double, Money))
305 destructor called. (temporary returned from operator*(double))
306 m1 > m5 is true  m2 > 1.23 is true  m4 == m6 is false
307 operator int() called
308 operator double() called
309 iv is 19  dv is 19.04
310 dollars of m2 is 19  cents of m2 is 4  total value of m2 is 19.04
311 And now enter a Money type: 4560.09
312 You entered $4560.09
313 destructor called. (m6)
314 destructor called. (m5)
315 destructor called. (m4)
316 destructor called. (m3)
317 destructor called. (m2)
318 destructor called. (m1)
Variable Table: Numbers in () are line numbers.

<table>
<thead>
<tr>
<th>319</th>
<th>m1</th>
<th>m2</th>
<th>m3</th>
<th>m4</th>
<th>m5</th>
<th>m6</th>
</tr>
</thead>
<tbody>
<tr>
<td>320</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>50</td>
<td>30</td>
</tr>
<tr>
<td>321</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>25</td>
<td>0</td>
<td>96</td>
</tr>
<tr>
<td>322</td>
<td>51</td>
<td>19</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>323</td>
<td>21</td>
<td>4</td>
<td>25</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>324</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>325</td>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>326</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>327</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>328</td>
<td>65</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>329</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>330</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>331</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>332</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>333</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>334</td>
<td>4560</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>335</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>336</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note:

337 252 m1 = 65; // Calls general constructor
338 252 // Syntax is used when only first argument is passed.
1. Recall that most operators can be overloaded as either a member function or a non-member function.

A non-member function has no invoking object. Thus, any objects, as well as other values needed by the function, are passed as arguments.

If an operator is overloaded as a non-member function, the left operand correspond to the first argument and the right operand correspond to the second argument.

2. Non-member functions cannot directly access private members of a class.

A function that is a friend of a class has the same access privileges as a member function, that is, a **friend function** can access private members.

3. **To create a friend function:**
   a. Place prototype, preceded by keyword **friend**, in class declaration

      A friend function is not a member function, but it has the same access privileges as a member function.

   b. Define the function.

      Do not use the class qualifier. It is not a member function.

      Do not use keyword **friend**.

4. In overloading an operator, use a friend function if the left operand is not an object of your class and the operation must directly access the data members of the class.
5. **Overloading `<<` operator:**

```cpp
Money investment;
cout << investment;  // output Money type like built-in types
```

`<<` is overloaded in the `ostream` class for each of the built-in types.

You could add another version of `operator<<( )` for `Money` type to the `ostream` class. Even if you were allowed to modify a standard class, it’s not a clever idea. It’s better to use the `Money` class.

Must be a friend since the lefthand operand is `cout`, not a `Money` object.

```cpp
operator<<(cout, investment);  // Call as friend
cout.operator<<(investment);  // Call as member function
```

Note:

a. The function must be a friend to the `Money` class since it accesses private members of a `Money` object, but it does not need to be a friend to the `ostream` class since it does not access private members of a `ostream` object.

b. The first argument can be any `ostream` object: `cout`, `cerr`, `clog`, or an `ofstream` object. Class `ofstream` is derived from class `ostream`.

c. Pass the stream by reference because the function must use the object itself, not a copy of the object.

d. The `Money` object could be passed by value or by reference. By reference is more efficient, both time-wise (no call to the copy constructor) and memory-wise (no copy of the object).

e. ```cpp
Money a, b;
cout << a << b;  // chained output operations
```

or

```cpp
operator<<(operator<<(cout, a), b);
```

The return value of the inner call is the lefthand argument to the outer call. This is the `ostream` object into which values are inserted. It should be the same `ostream` object that was used by the inner call.
operator<<() must return the ostream argument.

Since the ostream object is passed by reference to operator<<(), returning a reference to that argument has the effect of making the return value the object argument.

If operator<<() returned cout in line 202, then

```cpp
cerr << a << b;    // inner call
```

would insert a correctly into cerr, but b incorrectly into cout.

f. Returning an ostream object (not a reference return type) would return a copy of the ostream argument.

This means that

```cpp
cout << a << b;
```

would correctly insert a into cout, but b incorrectly into a copy of cout with unpredictable results.

Returning a reference type turns off the copying mechanism and returns the actual ostream object.

6. **Overloading >> operator:**

operator>>( ) has characteristics similar to operator<<():

a. Must be a friend since lefthand operand is istream object.

b. Lefthand argument can be any istream or ifstream type.

c. Pass streams by reference because the function must use the object itself, not a copy of the object.

d. The Money object could be passed by value or by reference. By reference is more efficient, both time-wise and memory-wise.

e. A reference to the istream argument must be the return value.
1. Review of conversions for built-in types:
   a. In assignment and initialization, if the type of the expression is not the same type as the receiving variable, the value of the expression is automatically converted to the type of the receiving variable, provided the two types are compatible.

   ```
   int iv = 5.987; // double 5.987 is converted to int 5
   double dv = 34; // int 34 is converted to double 34.0
   int* pi = 100; // int and int* are not compatible
   int* pi = (int*) 100; // valid
   ```

   b. If the type of an actual argument is not the same as the corresponding formal argument, a conversion to the type of the formal argument is done, provided the two types are compatible.

   ```
   void fun(double, int); // Prototype
   ...
   fun(2, 3.89); // fun(double (2), int (3.89));
   ```

   c. If the type of the return value is not the same as the type of the function, a conversion to the type of the function is done, provided the two types are compatible.

   ```
   int fun1()
   {
       return 3.89; // fun1() returns int (3.89)
   }
   ```

2. Conversion constructor: a constructor taking one argument converting from the type of the argument to the class type

   ```
   Money m6 = 30.96; // Valid - passing only
   Money m6 = Money(30.96); // Explicit call
   Money m6(30.96); // Implicit call
   Money m1;
   m1 = 65.0; // Implicit conversion
   // conversion constructor creates a temporary
   // Money object which is then assigned to m1
   ```

   Note that only a constructor having one argument can act as a conversion constructor.
3. `explicit Money(double d); // Allows only explicit conversions`  
   `// use explicit only in prototype`

   ```
   Money m6;
   m6 = 30.96; // Invalid - requires implicit conversion
   m6 = Money (30.96); // Valid - an explicit conversion
   ```

   If a conversion constructor is qualified with keyword `explicit`, it can only be used for explicit conversions with an explicit cast. Without keyword `explicit`, it can be used for both explicit and implicit conversions.

4. Implicit conversions are used when
   a. a `Money` object is initialized to a `double` type
   b. a `double` type is assigned to a `Money` object
   c. a `double` type is passed to a function that is expecting a `Money` argument
   d. a `Money` function returns a `double` type
   e. in a. thru d. above, a built-in type can be unambiguously converted to type `double`

   ```
   Money m6 = 20.F; // float 20.F is converted double 20.0 and
   // the conversion constructor is called
   ```

   If there were another conversion constructor that converts a `long double` to `Money`, then there would be the ambiguity of `float` to `double` or `float` to `long double` and the above statement would fail.

   Note: `Money m6 = 20.0; // calls conversion constructor`
   `Money m6 = 20; // calls general constructor`

5. `Money m6 = 30.96;`
   `Money m6(30.96);`
   `Money m6 = Money(30.96);`

   Note the above declaration statements are equivalent when you have the appropriate conversion constructor.
1. A conversion function does the opposite of a conversion constructor which converts from its argument type to a class type.

A conversion function converts from a class type to a built-in type. In other words, it is a technique that allows you to overload a type cast operator.

```
Money m1 = 24.35;
double dv1 = static_cast<double> (m1); // Explicit call
double dv2 = m1; // Implicit call
```

2. Syntax:

```
operator type();
```

Conversion function must:
- be a member function
- not specify a return type
  
  type specifies the type to which the class type is converted
- must have no arguments
  
  invoking object is converted

3. ```
   Money::operator int() const       // Line 165 in Money1Client
   {
      return dollars;
   }
   
   Money::operator double() const     // Line 171
   {
      return total_value();
   }
```  

Note that the proper type is returned even though there is no declared return type.

```
Money m1;
...
int iv = m1; // Implicit conversion
double dv = static_cast<double> (m1); // Explicit conversion
```
4.  

```cpp
   cout << m1;       // assumes no operator<<( )
```

Ambiguous: nothing indicates whether conversion should be to `int` or `double`. If there were only one of the conversion functions, there would be no ambiguity.

```cpp
   long lv = m1;
```

Ambiguous: either an `int` or a `double` can be assigned to a `long`. The compiler will not choose. If there were only the `operator int()` function, it would be used to convert a Money type to `int`, and then the `int` would be automatically promoted to `long`.

You can use an explicit type cast to indicate which conversion function to use.

```cpp
   long lv = static_cast<int> (m1);
```

5.  

Potential problems:
Defining a Money to double typecast implicitly defines all double-compatible operators for Money objects, whether appropriate or not.

```cpp
   Money m1;
   ...
   cout << sqrt(m1); // m1 would be converted to double
   // allowing you to take the square root
   // of a Money type
```

Generally it is better to use explicit conversions rather than implicit ones. Keyword `explicit` does not work with conversion functions

6.  

Solution:
Replace  
```cpp
   Money::operator int() { return dollars; }
```

With  
```cpp
   int Money::Money_to_int() { return dollars; } 
```

```cpp
   int iv = m1;  // replaced by  int iv = m1.Money_to_int();
```

Use caution with implicit conversion functions.

Too many conversion functions tend to create ambiguities.