This worksheet is for your use during and after lecture. It will not be collected or graded, but I think you will find it a useful tool as you learn C++ and study for the exams. Explain all false answers for the "True or False" questions; in general, show enough work and provide enough explanation so that this sheet is a useful pre-exam review. I will be happy to review your answers with you during office-hours, via Email, or instant messaging.

1. Use the following code listing to answer the questions beginning on page 1.

```
10 /**
11  * Wallet interface
12  */
13 class Wallet {
14 public:
15     // default ctor makes an empty, anonymous wallet
16     Wallet( );
17     Wallet( const string& theId);
18     Wallet( const string& theId,
19             int theDollars, int thePennies );
20     // retrieve the wallet identification
21     string identification( ) const;
22     // retrieve the monetary value
23     double getCash( ) const;
24     // track expenses
25     void expense( int costDollars, int costPennies );
26     void expense( double costAmount );
27     // track income
28     void income( int earnDollars, int earnPennies );
29     void income( double earnAmount );
30     // read value from the input stream --
31     // returns a reference to the istream argument provided
32     istream& input( istream& is );
33     // print myself to the output stream
34     // returns a reference to the ostream argument provided
35     ostream& output( ostream& os ) const;
36 private:
37     // makes sure monetary amounts are always positive,
38     // carries >= 100 pennies to dollars
39     // displays a message to cout if amount is negative
40     bool money_logic( int theDollars, int thePennies ) const;
41     // makes sure monetary amounts are positive
42     // displays a message to cout if amount is negative
43     bool money_logic( const double theAmount ) const;
44     // converts dollars and pennies to a decimal dollar amount
45     double dollars_pennies_to_value( int theDollars,
46             int thePennies ) const ;
47
48     // data members
49     string id; // owner name
50     double cash; // amount of money in wallet, >=0
51 };
```

(a) This is the class definition. True or false? Explain why your answer is correct.

Solution: False. This is a class declaration, also called the class interface. Add the class implementation to this declaration, and together they are the class definition.

(b) Why don't the function prototypes on lines 16 and 17 have return types?

Solution: They are the class constructors. Constructors are not allowed to have return types.

(c) How many member functions does `Wallet` have? How many data members?

Solution: Eleven (count the prototypes, thirteen if you count constructors), two data members (`id`, and `cash`).

(d) How many of `Wallet`'s member functions may change the calling object state?

Solution: Five, count the prototypes without a lingering `const;`. It would be nine if you count the constructors (which is OK). Technically, the object has yet to be created when the constructor is called, so whether or not it is "changed" is splitting hairs.

(e) Where will `id` and `cash` be initialized with values?

Solution: Data members should always be initialized in the class constructors.

2. Consider this implementation for `Wallet`'s three parameter constructor prototyped at line 19 on the preceding page.

```
3  void Wallet_3ctor( string& theId, double theCash, int theChange ) const
4  {
5      id = theId;
6      cash = theCash + double(theChange)/100;
7  }
```

State six things wrong with the constructor implementation.

Solution: It is missing the appropriate scoping: `Wallet::`, the first parameter should be a `const string&`, the second parameter should be an `int` type, a constructor can't be `const`, constructors cannot have return types, and constructors must be named the same as their class.

3. (a) Write the prototype for a `Wallet::get_dollars` function that returns the total value of bills in a `Wallet` object as an integer.

Solution: `int get_dollars( ) const;` or `int get_dollars( void ) const;`

(b) Now write the implementation for part a.

Solution: A straightforward implementation would be:

```
 7  double Wallet::get_dollars() const
 8  {
 9      // truncate off the cents
10      return int(value);
```

4. (a) Write the prototype for a `Wallet::clone` function: it returns a new `Wallet` object that is a duplicate of the calling object, **except** with a parameter specified new `id`.

Solution: `Wallet clone( const string& newId ) const;`

(b) Now write the implementation for part a.

Solution: A straightforward implementation would be:

```
10  Wallet Wallet::clone( const string& newName ) const
11  {
12      int dollars = int(value);  // truncate to dollars
13      int pennies = int((value-dollars)*100); // calc pennies
14      Wallet newWallet( newName, dollars, pennies );
15      return newWallet;
16  }
```

(c) Suppose a `main()` routine has created a `Wallet` object named `A`. Using the results of parts a and b, what C++ statement(s) can be added to `main` to create another `Wallet` object named `B` with the same amount of money as `A` but with a wallet `id` of `"CashFlow"`?

Solution: A one line solution would be:
`Wallet B = A.clone( "CashFlow");`

5. Suppose you have a C++ routine with an already created `Wallet` object named `aWallet` that was created with

```
Wallet aWallet( "Jackson", 20, 20 );
```

Use the implementation of `Wallet::output` shown below to answer parts a and b.

```
101 // print myself to the output stream
102 // returns a reference to the ostream argument provided
103 ostream& Wallet::output( ostream& os ) const
104 {
105     os << id << "␣wallet␣contains␣$" << cash;
106     return os;
107 }
```

(a) Write **two** C++ statements that would print `aWallet` to `cout` followed by an `endl`.

Solution: These two statements would do the trick:

```
aWallet.output( cout );
cout << endl;
```

(b) Now, write the C++ statement (note the singular, **one statement**) that would print `aWallet` to `cout` followed by the phrase "is not enough money.", and terminated with `endl`.

Solution: We can use the returned reference from the `output()` member function:

```
aWallet.output( cout ) << "␣is␣not␣enough␣money."<< endl;
```

(c) Write the C++ statement that would read **the output** of part a into `aWallet` if it were typed at the keyboard or read from a file.

Solution: `aWallet.input( cin );`

6. Suppose the `merger` function defined below belonged to the `Wallet` class.

```
24 /***
25  * Domestic "mergers" frequently result in the
26  * redistribution of funds as well as an agreed upon
27  * portion to be used as initial savings.
28  *
29  * In the end, money is simply moved around --- not
30  * created or consumed.
31  */
32 Wallet Wallet::merger( Wallet& w )
33 {
34     // remember these values
35     double totalValue( cash + w.cash );
36
37     // the new investment Wallet
38     Wallet cookieJar( "Investments" );
39
40     // divvy up the amounts
41     cash *= 0.5;
42     w.cash *= 0.75;
43
44     // redistribute funds --- no funds created or lost
45     cookieJar.cash = cash + w.cash;
46     cash = ( totalValue - cookieJar.cash ) / 2;
47     w.cash = ( totalValue - cookieJar.cash ) / 2;
48
49     return cookieJar;
50 }
51
52 int main()
53 {
54     Wallet Alice( "Alice", 80000, 0 ); // 79K
55     Wallet Bob( "Bob", 1200, 0 ); // 1.2K
56     Wallet Savings = Alice.merger( Bob );
57
58     Savings.output( cout ) << endl;
59     Alice.output( cout ) << endl;
60     Bob.output( cout ) << endl;
61     return 0;
62 }
```

(a) What will the `main` routine print as the value of each wallet after the `merger()` function is called?

Solution:

```
Investments $40900
Alice $20150
Bob $20150
```

(b) Would the result change if the collision had been `Savings = Bob.merger( Alice)`? Explain:

Solution: Yes, we expect it would change. `merger` takes money in different proportions from the calling object ($\frac{1}{2}$) as compared to the `w` parameter ($\frac{3}{4}$). So switching the wallets used for the calling object and parameter will *probably* give us different results. (If they were the same wallet, for instance `newW = Alice.merge( Alice );` then "switching" the calling object and parameter wouldn't make a difference.)

7. Consider the following true or false questions. You may simply identify a true statement as "true," and move on to the next question. **Rewrite any *false* statements** into a true statement about the same fundamental idea.

   (a) The `private` segment of a class should be used to hide secret or confidential information.

   > **Solution:** False. The `private` segment of a class should never have secret or confidential information stored in it without security-specific software practices. `private` has meaning only during compilation.

   (b) Constructor prototypes have a return type of `void`.

   > **Solution:** False. Constructor prototypes do not have a return type (they are one of two function types in C++ that don't; the other being *destructors*, which we don't study in this course).

   (c) Constructor prototypes may not have empty or `void` argument lists.

   > **Solution:** False. A constructor may have any type of argument list, including empty.

   (d) The compiler must see the class declaration (or *interface*) after the class definition.

   > **Solution:** False. Recall that *class declaration + class implementation = class definition*. To say the definition must be encountered (by the compiler) before either of its parts is nonsensical.

   (e) The `public` class segment is used for publishing data members to web sites.

   > **Solution:** False. The `public` class segment describes member functions and data members that may be accessed outside of "class scope".

   (f) A C++ *default constructor* accepts zero arguments.

   > **Solution:** True. In C++, the notion of a default constructor is a constructor with zero arguments.

   (g) The compiler provides all classes with a default constructor.

   > **Solution:** False. Only classes that do not define any constructors (there are other limitations as well, but I doubt we'll encounter them in this class).

   (h) "Helper" functions are `public` or `private` member functions that encapsulate chunks of class logic for reuse by application developers.

   > **Solution:** False. *Helper functions* are **private member functions** that help organize class specific logic and algorithms for the class developer.