

# C++ Classes

August 3, 2010

## The Players in the Class Solution

**The class Developer** The developer responsible for writing a class definition. *The whole point of writing a class definition is so an application developer will use it...*

**The Application Programmer** A programmer incorporating the class into an application (`main()`). Why use the class? It makes the job easier *if the class is well designed.*

**The End-User** Knowledge (or misknowledge) of the conceptual “object” dictated by the application developer.  
uncover<2->**Just wants the application to work the way it should.**

## The Players in the Class Solution

**The class Developer** The developer responsible for writing a class definition. *The whole point of writing a class definition is so an application developer will use it...*

**Must provide the application developer with enough “bang” to get the job done for the end user.**

**Not so much “bang” that a foot is blown off.**

**And write an easy to extend and maintain class definition.**

**The Application Programmer** A programmer incorporating the class into an application (`main()`). Why use the class? It makes the job easier *if the class is well designed.*

**Just wants the class to work the way it should.**

**The End-User** Knowledge (or misknowledge) of the conceptual “object” dictated by the application developer.

**Just wants the application to work the way it should.**

# Classy Knowledge

	Member Data
	Member Functions
Class & main() Anatomy	Accessor Functions
Multiple Source File Builds	Helper Functions
Class Scope	static const Data Members
Class <i>ctor</i> & <i>dtor</i>	Array Data Members
Class “Protections”	Array’s of Class Objects

View Wallet\_main.cxx

## Class & main() Anatomy

Three parts:

1. The **Class Declaration** (or **Interface**)

Pretty much everything inside of the **header file** `Wallet.h`.

2. The **Class Implementation**

Pretty much everything inside of `Wallet.cxx`.

3. The `main()` Routine

**Class Definition** = **Class Declaration** + **Class Implementation**

analogous to

**Function Definition**  $\approx$  **Function Prototype** + **Function Header & Body**

# Arranging the Parts

## One main.cpp File

```
class declaration;
class implementation;

int main()
{
    :
}
```

main() at the End

## One main.cpp File

```
class declaration;

int main()
{
    :
}

class implementation;
```

main() Stuck in the Middle

## Multiple Source Files

### Declaration File classname.h

```
class declaration;
```

### Implementation File classname.cpp

```
#include "classname.h"

class implementation;
```

### main.cpp File

```
#include "classname.h"
int main()
{
    :
}
```

# Class Declaration

```

13 class Wallet {
14 public:
15     // default ctor makes an empty, anonymous wallet
16     Wallet( );
17     Wallet( const string& theName );
18     Wallet( const string& theName ,
19             int theDollars , int thePennies );
20     // retrieve the wallet identification
21     string id( ) const;
22     // retrieve the monetary value
23     double getValue( ) 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;

```

- ▶ Notice the `#pragma` once at line 4 in `Wallet.h`.  
Non-portable solution, but adequate for this course.
- ▶ Notice the `class Name{...};` wrapper.
- ▶ `public` & `private` postponed...

# Class Declaration

```

13 class Wallet {
14 public:
15     // default ctor makes an empty, anonymous wallet
16     Wallet( );
17     Wallet( const string& theName );
18     Wallet( const string& theName ,
19             int theDollars , int thePennies );
20     // retrieve the wallet identification
21     string id( ) const;
22     // retrieve the monetary value
23     double getValue( ) 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;

```

- ▶ Strange “ctors” look like function prototypes but lack a return type.
- ▶ The other function elements (class methods) look like normal function prototypes.

## Class Declaration

```
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 name; // owner name
50     double value; // amount of money in wallet , >=0
51 };
```

- ▶ The data elements look like normal variable declarations.  
**Garbage Values?**

# Class Implementation

```
1 /**
2  * A Wallet class
3  */
4
5 #include <iostream>      // system headers first
6 #include <string>
7 #include "Wallet.h"    // developer headers second
8 #include "string_substitute.h" // replace whitespace with _
9 using namespace std;
10
11 /**
12  * Wallet implementation
13  */
14
15 // default ctor makes a anonymous, empty wallet
16 Wallet::Wallet( )
17 {
18     name = "???" ; // anonymous marker
19     value = 0;
20 }
```

- ▶ We use `#include "..."` to grab the class declaration.  
**The "`"` vs. `<>` means: look for the header file in the same folder or directory as this source file.**
- ▶ Scope resolution operators `::` before function names.

# Class Implementation

```
1 /**
2  * A Wallet class
3  */
4
5 #include <iostream>      // system headers first
6 #include <string>
7 #include "Wallet.h"     // developer headers second
8 #include "string_substitute.h" // replace whitespace with _
9 using namespace std;
10
11 /**
12  * Wallet implementation
13  */
14
15 // default ctor makes a anonymous, empty wallet
16 Wallet::Wallet( )
17 {
18     name = "???" ; // anonymous marker
19     value = 0;
20 }
```

- ▶ These functions have a special *class scope*: They see data elements defined in the class declaration!
- ▶ Otherwise, these are normal looking function definitions!

Which file contains the functions written in Wallet's class scope?

## What is Class Scope?

Within member function definitions, class data members belong to the **calling instance** or **calling object**.

Wallet Implementation of `Wallet::getValue()`

```
49 // retrieve the monetary value
50 double Wallet::getValue( ) const
51 {
52     return value;
53 }
```

Application (Driver) Code uses `Wallet::getValue()`

```
37 cout << "Bob tries to spend 10 dollars..." << endl;
38 // can't be done, Bob is broke
39 Bob.expense( 10, 0 ); // generates error message, Bob is broke
40 Alice.expense( 10.0 ); // this is OK, Alice has cash
41 outfile << "Alice spends 10 dollars, her new value is:" <<
42     Alice.getValue() << endl;
43 dualOutput( Bob, cout, outfile );
```

How does  
`getValue()` know  
the amount to  
display?

## Constructors and Destructor

**Constructor** (*ctor*) A class constructor is called every time an instance variable is created.

**Only one constructor is called**, even if a class has multiple constructors defined.

**Destructor** (*dtor*) The class destructor is called whenever a class instance's lifetime expires.

In the Wallet application code, the Wallet destructor for anonymous, Bob, and Alice is called immediately before `return 0;` is executed. **The compiler does this automatically.** Unlike ctors, there may be only one destructor defined for a class.

We won't study destructors in detail for this course.

## The Constructor's Job

A class constructor must initialize all data members so that the class instance represents a “consistent” state.

**SO THAT** other class member functions may **assume** that the internal state of every class instance is in a “consistent” state.

```

23 Wallet::Wallet( const string& theName )
24 {
25     name = string_substitute( theName, "\n\t", '_' );
26     value = 0;
27 }
28
29 Wallet::Wallet( const string& theName,
30               int theDollars, int thePennies )
31 {
32     name = string_substitute( theName, "\n\t", '_' );
33     value = 0;
34     if( money_logic( theDollars, thePennies ) ) {
35         // valid amount
36         value = dollars_pennies_to_value( theDollars,
37                                           thePennies);
38     }
39     // otherwise, money logic has displayed an
40     // error message

```

What might an inconsistent state of Wallet be?

Note the different names for parameters and data members.

## Constructor Details

- ▶ The name of a constructor must be the class name.
- ▶ The **default constructor** accepts no arguments.
- ▶ A constructor cannot have a return type, not even `void`.
- ▶ If a constructor is not provided for a class, then the compiler automatically provides a default constructor.  
The compiler-supplied default constructor simply initializes each **class** data element before returning.  
**It does not initialize fundamental data type members!**
- ▶ If you provide *any* constructors with your class, then the compiler does nothing. → **You must write your own default constructor.**

Does `Wallet` have a default constructor?

## public & private

`public` These class functions or data members may be accessed by the *any* programmer via the "." operator.

Note the use of `w.output( out1 );` on line 16 in the driver program, and `Bob.expense(10,0);` on line 39.

`private` The *compiler* prevents these class functions or data members from being accessed *except* for source in the class scope (specifically class functions).

Which file contains code that calls  
**only** public functions of the `Wallet` class?

## “private” to the Compiler Only

private data members and functions allow “expressiveness” in the C++ language. It is how Class Developers say “hands-off” to Application Developers.

private data and functions are easily examined and called by a programmer that simply puts their mind to doing so.

## Data Members

**Data Member** Any variable or reference in a class.

Examples: name, value in Wallet.

- ▶ Declare them precisely as you would (uninitialized) variables or references within a function's local scope.  
**Where do they get initialized?**
- ▶ There are other qualifiers for data member declarations, but they won't be covered in this course. (Will your project need them? Maybe, see me if you hit a big, hard wall.)

**What's the difference between a public and private data member?**

## Data Members

In general, non-constant data members should **ALWAYS** be private.

Have a thorough explanation (in source comments) for non-constant public data members.

## Member Functions

**Member Function** Any function prototyped in a class declaration that requires the scope resolution operator (`::`) in its implementation.

- ▶ Declare them as you do function prototypes (*function header + ;*)
- ▶ Use a **const qualifier at the end** to signify that the member function doesn't change the state (or "value") of the calling object. In which case, the const qualifier must be provided in the function implementation as well. The const goes between the closing `)` and the `;`.

How many member functions does `Wallet` have?

How many change the value of `Wallet`'s data members?

## Accessor Functions

If all the pertinent data representing a class is private, how does an Application Programmer change them?

Through Class Developer provided *accessor functions*!

## Accessor Functions

**Accessor Function** A **public** member function that allows the value of an internal private data member to be either retrieved or changed.

Typically prototyped as:

```
/**
 * Accessor functions for an integer foo.
 */
class class_name {
private:
    int foo;
public:
    int get_foo() const;
    void set_foo( int newfoo );
};
```

```
/**
 * Accessor Functions for an integer foo.
 */
class ClassName {
private:
    int foo;
public:
    int getFoo( void ) const;
    void setFoo( int newFoo );
};
```

If you follow one of these common naming schemes, you don't need a function header comment for accessor functions.

# Accessor Functions

```
/**  
 * Accessor functions for an integer foo.  
 */  
class class_name {  
    private:  
        int foo;  
    public:  
        int get_foo() const;  
        void set_foo( int newfoo );  
};
```

```
/**  
 * Accessor Functions for an integer foo.  
 */  
class ClassName {  
    private:  
        int foo;  
    public:  
        int getFoo( void ) const;  
        void setFoo( int newFoo );  
};
```

Why is the GET const?

When do you use private accessor functions?

How many accessor functions does Wallet have?

# Accessor Functions

```
/**  
 * Accessor functions for an integer foo.  
 */  
class class_name {  
    private:  
        int foo;  
    public:  
        int get_foo() const;  
        void set_foo( int newfoo );  
};
```

```
/**  
 * Accessor Functions for an integer foo.  
 */  
class ClassName {  
    private:  
        int foo;  
    public:  
        int getFoo( void ) const;  
        void setFoo( int newFoo );  
};
```

Does the Wallet class have “complete” accessors (a getter and a setter for each data member)?

# Accessor Functions

```
/**  
 * Accessor functions for an integer foo.  
 */  
class class_name {  
    private:  
        int foo;  
    public:  
        int get_foo() const;  
        void set_foo( int newfoo );  
};
```

```
/**  
 * Accessor Functions for an integer foo.  
 */  
class ClassName {  
    private:  
        int foo;  
    public:  
        int getFoo( void ) const;  
        void setFoo( int newFoo );  
};
```

Does the Wallet class have “complete” accessors (a getter and a setter for each data member)?

What would they look like?

## “Helper” Functions

“Helper” Function A **private** member function representing chunks of logic used by other (public or private) member functions.

How many helper functions does `Wallet` have?

## Providing OO Input & Output

Any reasonably mature object should have routines that allow reading and writing of its value with input and output streams (cin, cout, **and file streams**).

This means the input routine does **NOT** display prompt to the user!

This (usually) means the output routine does **NOT** use endl or '\n'!

## Providing OO Input & Output

### Wallet.h Header File

```
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;
```

- ▶ The names `input` and `output` seem the most symmetric pairing, but other names are common (`read & write`, `input & print`).
- ▶ Note the output function is `const` (**why?**) and that the streams are passed by reference.

## Providing OO Input & Output

### Wallet.cxx Implementation File

```
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 << name << "_wallet_contains_" << value;
106     return os;
107 }
```

- ▶ Return the stream reference passed to function!
- ▶ Use ostream& and istream& types, this allows the same output and input routines to be used for both console and disk files!

# Providing OO Input & Output

## Wallet.cxx Implementation File

```

109 // read value from the input stream —
110 // returns a reference to the istream argument provided
111 istream& Wallet::input( istream& is )
112 {
113     // read into local vars first
114     string n;
115     double v;
116     string wallet, contains;
117     char dsign;
118     if((is>>n>>wallet>>contains>>dsign>>v) && (dsign=='$')) {
119         // make sure money amount is ok
120         if( money_logic( v ) ) {
121             // aok — store data in object
122             name = n;
123             value = v;
124         } else {
125             // error! put the input stream into an error state
126             is.setstate( ios::failbit );
127         }
128     }
129     // return the input stream parameter, if the input
130     // statement failed, is is already in an error state
131     return is;
132 }

```

- ▶ Note the care taken inside of input to assure that a valid value is read, and that the fail() state of the input stream reflects the success of the operation.
- ▶ Use setstate with ios::failbit to signal failure

# Providing OO Input & Output

The **input** member function of a class consists of three steps:

```

109 // read value from the input stream —
110 // returns a reference to the istream argument provided
111 istream& Wallet::input( istream& is )
112 {
113     // read into local vars first
114     string n;
115     double v;
116     string wallet , contains;
117     char dsign;
118     if((is>>n>>wallet>>contains>>dsign>>v) && (dsign=='$')) {
119         // make sure money amount is ok
120         if( money_logic( v ) ) {
121             // aok — store data in object
122             name = n;
123             value = v;
124         } else {
125             // error! put the input stream into an error state
126             is.setstate( ios::failbit );
127         }
128     }
129     // return the input stream parameter, if the input
130     // statement failed, is is already in an error state
131     return is;
132 }

```

- Read data values into local variables,
- validate the data (is it logical for a new object's state?),
- and store the new values into the object data members **IF** they are OK.

## The point Class

constant Data Members

Arrays as Data Members

Anonymous Objects

Arrays of Objects

[Edit Class](#)

[Edit Driver](#)

[Run Driver](#)

## Declaring Class Constants

```

point.h
14 public:
15     // compiled as a three dimensional
16     // point
17     static const int COORDS = 3;
18     // for input and output notation
19     static const char OPENTUPLE = '<';
20     static const char CLOSETUPLE = '>';
21     // default ctor makes "origin"
22     point ( );
23     // origin by another name
24     point( const string& theName );

point.cxx
189 ostream& point::output( ostream& os ) const
190 {
191     // prints Blah blah blah < X Y Z >
192     os << name << "_" << OPENTUPLE;
193     for( int i=0; i<COORDS; i++ ) {
194         os << "_" << coords[i];
195     }
196     os << "_" << CLOSETUPLE;
197     return os;
198 }

```

Best way to declare constant data needed by all class instances.

Works for fundamental INTEGRAL data types only.

Must be initialized in class declaration.

## Arrays as Data members

```
point.h
14 public:
15     // compiled as a three dimensional
16     // point
17     static const int COORDS = 3;
18     // for input and output notation
19     static const char OPENTUPLE = '<';
20     static const char CLOSETUPLE = '>';
21     // default ctor makes "origin"
22     point ( );
23     // origin by another name
24     point( const string& theName );

56 private:
57     // helper functions
58     void init_values( double x );
59     void init_values( const double xi[], const int num );
60     // data members
61     string name;
62     double coords[COORDS];
63     };
```

Like fundamental data types as data members, arrays are declared **without** initial values.

## Arrays as Data members

```
point.cxx
10 // origin maker
11 point::point ( )
12 {
13     init_values( 0 );
14     name = "origin";
15 }
16
17 point::point ( const string& theName )
18 {
19     name = theName;
20     init_values( 0 );
21 }
22
23 // arbitrary point maker
24 point::point( const string& theName ,
25             const double xi[], const int num )
26 {
27     name = theName;
28     init_values( xi , num );
29 }
```

Each constructor must initialize **each** element of **each** array data member.

## Arrays as Data members

point.cxx

```

17 point::point ( const string& theName )
18 {
19     name = theName;
20     init_values( 0 );
21 }
22
23 // arbitrary point maker
24 point::point( const string& theName ,
25             const double xi[], const int num )
26 {
27     name = theName;
28     init_values( xi , num );
29 }

```

point.cxx

```

200 // helper — init by array
201 void point::init_values( const double xi[], const int num )
202 {
203     // either num or COORDS will stop the loop
204     int i=0;
205     for( ; i<num && i<COORDS; i++ ) {
206         coords[i] = xi[i];
207     }
208     // fill any unspecified elements with zero
209     for( ; i<COORDS; i++ ) coords[i] = 0;
210 }

```

Each constructor must initialize **each** element of **each** array data member.

init\_values is a **helper function** supporting DRY in the ctors.

## Arrays as Data members

point.h

```
28 // accessor functions
29 void setName( const string& newname );
30 string getName( ) const;
31 void setCoord( int coord, double newvalue );
32 double getCoord( int coord ) const;
```

point.cxx

```
42 void point::setCoord( int coord, double newvalue )
43 {
44     // be sure to stay within array size with % COORDS
45     coords[coord%COORDS] = newvalue;
46 }
47
48 double point::getCoord( int coord ) const
49 {
50     // be sure to stay within array size with % COORDS
51     return coords[coord%COORDS];
52 }
```

Accessor functions for array data members (usually) require a second parameter representing the **index** to **set** or **get**.

## Classname ( ) Anonymous Objects

point.cxx

```

54 // reflection through origin
55 // returns a NEW point object
56 point point::reflect() const
57 {
58     double reflected_coords[COORDS];
59     for( int i=0; i<COORDS; i++ ) {
60         reflected_coords[i] = -coords[i];
61     }
62     // returns an "anonymous" object
63     return point( name + "_reflected",
64                 reflected_coords, COORDS );
65 }

```

```

82 point point::negate( ) const
83 {
84     point neg = reflect(); // same as reflection
85     // but we give it a more mathy name
86     // NOTE two anonymous string objects in calculation
87     neg.name = string( "-" ) + name + string( " " );
88     return neg;
89 }

```

Creation of temporary objects to aid in equations or for return values from functions.

## Accessing Class Scope Parameters

point.cxx

```
67 // Simple operations for points —
68 // Each returns a NEW point object
69 point point::add( const point& p ) const
70 {
71     // create a point with appropriate name, and calculate
72     // the coordinate sum
73     point sum( name + " + " + p.name );
74     for( int i=0; i<COORDS; i++ ) {
75         sum.coords[i] = coords[i] + p.coords[i];
76     }
77
78     return sum;
79
80 }
```

A class scoped function has access to the private data members and functions of any **parameter** of the same class.

## Arrays of Objects in `main()`

[View Data](#)

[View Path](#)

[Edit Source](#)

[Run Program](#)

fnis

-