The development of a CreditCard class

(an introduction to object-oriented design in C++)

January 20, 2006

Rather than explain the many aspects involved in a fully-fledged, yet complex example, we will build up to such an example in a piecewise fashion. Thus we start with a relatively simple program and add complexity as we go.

Version 1

In Figure 1, we give a self-contained file which defines a CreditCard class and then shows a sample main routine which utilizes that class. Already, there are many aspects of this class definition which deserve comment. Lines 12–14 are used to bring in some definitions from standard libraries to support basic input/output functionality and a defined **string** class. Though we might think that these basic needs would be part of the core language, they are not. Instead they are part of a very large extended library that is relatively standard across all installations of the language.

The official declaration of the CreditCard class begins at line 16 and continues up to the closing brace at line 45. Lines 18–21 formally declare the data members of this class, including explicit type declarations (e.g. **string**, **int**, **double**).

The next section is used to define the member functions. Please note that there was no explicit requirement to declare the data members separately from the member functions; they can be interspersed as we'd like, though certainly grouping them helps the legibility. In our first version of the code, we define only three member functions, CreditCard, chargeIt, and makePayment. Notice that the signatures of each method (i.e., lines 25, 33, 42) follow a very specific general format, explicitly declaring the type of any parameter as well as the type of return value. For example, the chargeIt method takes one parameter, which is a double-precision floating point number, and in the end returns a boolean value. A few special cases to note. The makePayment body does not explicitly return anything. In this case, the return type is declared using a special keyword, **void**, which designates this routine as one which does not have any return value (we informally describe this as a "**void** return type").

The declaration of the CreditCard method is special in its own right. This method serves as a constructor for the class. Though it is common to list this first, this is not actually a requirement. The way that the compiler recognizes this as a constructor is because the method name is the same as the class name. A second special feature of this signature, versus

```
12 #include <iostream>     // provides input/output classes
13 #include <string>       // provides string
14 using namespace std;    // make the included definitions accessible
15
16 class CreditCard {
17   //----------------------  data members ----------------------
18   string     number;     // credit card number
19   string     name;       // card owner's name
20   int        limit;      // credit limit
21   double     balance;    // credit card balance
22
23   //----------------------  member functions ----------------------
24   // --- constructor ---
25   CreditCard(string no, string nm, int lim, double bal=0) {
26     number = no;
27     name = nm;
28     balance = bal;
29     limit = lim;
30   }
31
32   // --- update methods ---
33   bool chargeIt(double price) {     // make a charge
34     if (price + balance > limit)
35       return false;                 // over limit
36     else {
37       balance += price;
38       return true;                  // the charge goes through
39     }
40   }
41
42   void makePayment(double payment) {     // make a payment
43     balance -= payment;
44   }
45 }; // end of CreditCard definition
46
47 int main() {
48   CreditCard discover("6011 1234 5678 9999","Michael",5000);
49   cout << "current balance is " << discover.balance << "\n";
50
51   discover.chargeIt(400.00);
52   cout << "current balance is " << discover.balance << "\n";
53
54   discover.makePayment(100.00);
55   cout << "current balance is " << discover.balance << "\n";
56 }
```

Figure 1: Excerpt from Version 1 of the CreditCard program (CreditCard.cpp)

the standard methods, is that there is nothing listed at all in the place where a return type specifier would normally be placed (not even **void** is used). In essence, the constructor is generally used to initialize the state of a newly created instance. There is no explicit return value in this context.

One other important point is to notice how the data members of an instance are accessed from within the body of the methods. For example, note the use of balance in the chargeIt method. Recall that all identifiers must be formally declared, yet this is not declared, at least locally in the context of that method body. If it cannot find such a local declaration, it then looks to see if a data member exists with that name (which it does in this case). Since variables are not always explicitly declared in Python, the way to distinguish between local variables and data members is by designating use of data members prefaced with an explicit self reference. There actually is an implicit reference to the instance of an object in C++ as well, designated by the implicit keyword, **this**. However you are not required to use **this** when accessing members of the instance.

Hopefully, most of this example is self-explanatory. If we wish to run this code, we must first compile it into an ***executable***. Since the entire program is in a single source file, named `CreditCard.cpp`, we can execute the following command to the operating system:

```
g++ -o CreditCard CreditCard.cpp
```

The '-o CreditCard' flag designated that we want the generated executable to be a file named `CreditCard` (the default is otherwise `a.out`). Unfortunately, when we attempt to compile this code, we run into a problem. We see a warning:

```
CreditCard.cpp:16: warning: all member functions in class ‘CreditCard’ are
   private
CreditCard.cpp: In function ‘int main()’:
CreditCard.cpp:25: error: ‘CreditCard::CreditCard(std::basic_string<char,
   std::char_traits<char>, std::allocator<char> >, std::basic_string<char,
   std::char_traits<char>, std::allocator<char> >, int, double)’ is private
CreditCard.cpp:48: error: within this context
```

The problem is as follows. C++ takes the principle of encapsulation very seriously and provides many means for specifying what functionality is to be accessible for varying roles in a larger software project. We refer to this as ***access control***. In particular, each individual member of a class (both data and function) have a designated access mode, which is one of the following.

- **private** — Such a member of an object can only be accessed from within the implementation of the class itself (e.g., from the body of a member of this class)
- **protected** — Such a member of an object can be accessed from within the implementation of the class itself, or from within the implementation of any child class which is defined based upon this class (only relevant when using inheritance).
- **public** — Such a member of an object can be accessed from anywhere within the larger software project.

Furthermore, if we fail to specify an access mode within a class definition, the default behavior is that all members are **private**. So as a **private** member, it is perfectly fine that balance is accessed from within the body of the chargeIt routine, but it is unacceptable that balance is accessed at line 49 because this line is not within the scope of the class definition itself. In fact, even the constructor is private, so we cannot legally instantiate an object of this class.

Version 2

As an initial remedy to the problem of default **private** status of members, our next version of the code explicitly declares the desired access mode. This is done by using the designation of **public**:, **protected**:, or **private** prior to the declaration of a member. In fact, once such a designation is made, that access mode is used for all of the following declared members, unless we later re-designate the desired access mode.

We present an almost identical class definition in Figure 2. The only change in that definition is the designation of **public**: access mode at what is now line 14 (our line numbers change from version to version because of some preface documentation we are not including in this handout) Therefore, all members in this version are **public**. This remedies the original hurdle we had, and our program can now be compiled and then executed.

But there is a potential problem with the convention of leaving all members publicly accessible. If we take a very defensive posture as to the integrity of our class, we might be concerned about how users of our class will interact with our objects. If those users follow the expectations, everything goes well. But there is also room for abuse. In the revised main routine we give an example of proper use of a CreditCard object, as well as improper use. Though we have no concern with someone looking at the current value of the balance member, such as at line 47, we might be worried about letting someone arbitrarily change the value of balance, as done at line 56. Unfortunately, with that data member designated with **public** access control, we really cannot stop such modifications.

Version 3

To remedy the problem in the previous version, we wish to tighten the access mode as much as possible, while still leaving a public user of our class with sufficient functionality to suit their needs. As a general rule, we will never want to have the data members themselves declared to be **public**. If Figures 3–4, we give our latest version of the code. In this case, we designate the data members as **private** at line 15. Declaring those data members as **private** stops an outside piece of code from modify the value of that data. Yet it also stops them from even reading the value.

Because we often want to provide a public way to read data, yet without allowing direct manipulation, we introduce what we term ***accessor methods***. For example, we have balance as a **private** data member, yet we offer a **public** method named, getBalance() which returns the current value of that variable. Notice that line 46 is perfectly legal because that code is being written within the scope of the class definition itself. Line 66 is legal because the method being called is indeed a **public** one. Notice that the direct manipulation in

```
 9 #include <iostream>     // provides input/output classes
10 #include <string>       // provides string
11 using namespace std;    // make the included definitions accessible
12
13 class CreditCard {
14 public:
15   //----------------------  data members ----------------------
16   string     number;     // credit card number
17   string     name;       // card owner's name
18   int        limit;      // credit limit
19   double     balance;    // credit card balance
20
21   //----------------------  member functions ----------------------
22   // --- constructor ---
23   CreditCard(string no, string nm, int lim, double bal=0) {
24     number = no;
25     name = nm;
26     balance = bal;
27     limit = lim;
28   }
29
30   // --- update methods ---
31   bool chargeIt(double price) {     // make a charge
32     if (price + balance > limit)
33       return false;                 // over limit
34     else {
35       balance += price;
36       return true;                  // the charge goes through
37     }
38   }
39
40   void makePayment(double payment) {     // make a payment
41     balance -= payment;
42   }
43 }; // end of CreditCard definition
44
45 int main() {
46   CreditCard discover("6011 1234 5678 9999","Michael",5000);
47   cout << "current balance is " << discover.balance << "\n";
48
49   // proper use
50   discover.chargeIt(400.00);
51   cout << "current balance is " << discover.balance << "\n";
52   discover.makePayment(100.00);
53   cout << "current balance is " << discover.balance << "\n";
54
55   // credit card fraud
56   discover.balance -= 200.00;
57   cout << "current balance is " << discover.balance << "\n";
58 }
```


Figure 2: Excerpt from Version 2 of the CreditCard program (CreditCard.cpp)

```
10 #include <iostream>     // provides input/output classes
11 #include <string>       // provides string
12 using namespace std;    // make the included definitions accessible
13
14 class CreditCard {
15 private:
16   //----------------------  data members ----------------------
17   string     number;     // credit card number
18   string     name;       // card owner's name
19   int        limit;      // credit limit
20   double     balance;    // credit card balance
21
22 public:
23   //----------------------  member functions ----------------------
24   // --- constructor ---
25   CreditCard(string no, string nm, int lim, double bal=0) {
26     number = no;
27     name = nm;
28     balance = bal;
29     limit = lim;
30   }
31
32   // --- accessor methods ---
33   string getNumber() {
34     return number;
35   }
36
37   string getName() {
38     return name;
39   }
40
41   int getLimit() {
42     return limit;
43   }
44
45   double getBalance() {
46     return balance;
47   }
```

Figure 3: Version 3 of the CreditCard program (CreditCard.cpp); continued in Figure 4

line 75 is illegal, and in fact unless we remove that line, the compilation of this code will fail with the following error message:

```
CreditCard.cpp: In function 'int main()':
CreditCard.cpp:20: error: 'double CreditCard::balance' is private
CreditCard.cpp:75: error: within this context
```

```
49    // --- update methods ---
50    bool chargeIt(double price) {     // make a charge
51      if (price + balance > limit)
52        return false;                 // over limit
53      else {
54        balance += price;
55        return true;                  // the charge goes through
56      }
57    }
58
59    void makePayment(double payment) {     // make a payment
60      balance -= payment;
61    }
62  }; // end of CreditCard definition
63
64  int main() {
65    CreditCard discover("6011 1234 5678 9999","Michael",5000);
66    cout << "current balance is " << discover.getBalance() << "\n";
67
68    // proper use
69    discover.chargeIt(400.00);
70    cout << "current balance is " << discover.getBalance() << "\n";
71    discover.makePayment(100.00);
72    cout << "current balance is " << discover.getBalance() << "\n";
73
74    // credit card fraud impossible
75    discover.balance -= 200.00;
76  }
```

Figure 4: Continuation of Version 3 of the CreditCard program (CreditCard.cpp)

Version 4

The accessor methods in our previous version were sufficient to allow a user to query any individual data member value. However for convenience, we might wish to be able to get a string representation of the entire state of the card. For this purpose, we add a toString() method to our implementation. Unfortunately, there is a bit of ugliness at this point because of our need to convert the basic data types such as **int** and **double** to string representations. One might think that this would be a standard task and fully supported by the language, but oddly it is not quite so easy. We have provided a section of code to provide a generic ToString conversion function (note our use of an uppercase T in naming this method). Based upon that, we can now provide our own custom, toString method of our CreditCard class (note our use of a lowercase T in naming this method). That particular method is declared as follows.

```
string toString() {
  string result;
  result =
    "Number = "  + number + "\n" +
    "Name = "    + name + "\n" +
    "Balance = " + ToString(balance) + "\n" +
    "Limit = "   + ToString(limit) +  "\n";
  return result;
}
```

We may now use this newly defined **public** method from within our main routine, as shown in the following example.

```
CreditCard discover("6011 1234 5678 9999","Michael",5000);
cout << discover.toString() << "\n";
```

If we were to run this portion of the code, the generated output reads as,

```
Number = 6011 1234 5678 9999
Name = Michael
Balance = 0
Limit = 5000
```

Version 5

Taking our previous improvement a step further, it becomes even more convenient for an end user to be able to syntactically output a CreditCard object directly, without explicitly having to call our toString function. That is, what if we wished to replace the previous syntax

```
CreditCard discover("6011 1234 5678 9999","Michael",5000);
cout << discover.toString() << "\n";
```

with the even simpler syntax

```
CreditCard discover("6011 1234 5678 9999","Michael",5000);
cout << discover << "\n";    // We no longer call toString()
```

Our reason for supporting this syntax is purely for the convenience of the end user of our class, but after all, that is the whole point of us defining a class in the first place.

In order to add this support, we formally need to do something which is called ***overloading an operator***. In this case, the operator in question, `<<`, is the output stream operator. That is, an expression such as (a << b) is formally evaluated depending upon the type of the operands a and b. The syntax used for overloading an operator is one which is similar in style to a function definition. In this case, our signature appears as follows.

```
ostream& operator<<(ostream& out, CreditCard& c) {
```

Rather than specifying a name of a new function, we designate that we wish to overload a particular operator, in this case **operator**<<. We specify the two operands of this binary operator as if they were parameters to a function. That is in this case, we are concerned with the behavior when faced with an expression of the form (*ostream* << *CreditCard*). The resulting type of the expression is specified in the signature as if it were a return value of a function. Please note that we are temporarily ignoring the issue of why ampersands (&) are used in this particular signature. We will discuss that another time. A more detailed discussion of operator overloading is given in the text as well.

Version 6

In our second-to-last revision of the class, we introduce another form of access control via a keyword, **const**. In many different forms, there are times in our program where we may wish to declare an entire object to in essence be "read-only" for some period of time.

For example, in our new version of the main routine, we instantiate a second CreditCard as follows:

```
const CreditCard visa("5391 0375 9387 5309","Michael",10000);

// acessors are allowed
cout << "Balance is " << visa.getBalance() << "\n";

// mutators are not
visa.chargeIt(500.00);
```

The use of the keyword, **const**, in the declaration of visa informs the compiler that we will never again change the state of that object, once it has been constructed.

The compiler, in turn, will subsequently ***enforce*** this condition! However to do so, it might suggest that the compiler needs to trace through the logic of our program. For example, if we call some method, visa.foo(), it is not immediately evident whether such a method changes the state of the object. The compiler could look into the body of that method, but that may not be immediately available and worse yet, even looking at that body may not definitively determine whether the object is changed. For example if the implementation of that method involves a calls to another method, then the compiler would have to determine the behavior of that method, and so on.

The solution, from the compiler's perspective, is that it requires that the programmer explicit label as safe any methods which are guaranteed not to modify the state of the given object. That is, in our credit card context, we will explicitly mark the getBalance method as one which does not modify the object (i.e., an accessor); in contrast the chargeIt method is a mutator and so it will not be marked as safe. Therefore, we replace the original definition of the getBalance method, which appeared as

```
double getBalance() {
  return balance;
}
```

with the new signature,

```
double getBalance() const {
  return balance;
}
```

Notice the addition of the keyword, **const**, to the right side of the method declaration though before the actual body. In this context, this declaration guarantees that calling this method will not change the state of the object upon which it is invoked. In this example, we will not add this tag to the chargeIt method because that method does intentionally change the state of the object. By tagging our accessor methods in this fashion, we will then be allowed to call them on those 'const' declared objects, such as visa.getBalance().

Furthermore, once we declare a method to be a 'const' method, the compiler enforces this condition upon us. That is, if the body of such a method makes a statement which may change the state of the object, this results in an error at compile time.

There are other contexts in which we will use the **const** keyword at times. In fact, we already subtlety slipped one such use into an earlier example without explanation at the time. If you look a the signature of the overloaded output operator

```
std::ostream& operator<<(std::ostream& out, const CreditCard& c) {
  out << c.toString();
  return out;
}
```

you will notice the use of that word immediately in front of the declaration of the second parameter. In this context, it offers a guarantee that the object which is being sent as a parameter will not be altered by such a call.

Version 7

In our final example for this lesson, our goal is to transition from our earlier examples which are self-contained in a single file, to demonstrate how the source code for a program can be broken across many files. Though unnecessary for such a small example, the ability to break apart a program across files becomes greatly significant as we consider the development of larger and larger systems.

If multiple people are working on different components of a larger system, it becomes convenient to edit different files rather than to simultaneously edit the same file. As another benefit, if we envision some of our classes to be reusable in many different applications, we would like to be able to maintain the code for that class independently, rather than to have to cut-and-paste that code separately into two or more different projects. By having one centralized implementation, it becomes easier to fix bugs or make improvements universally.

With our credit card example, we will break the program into three distinct files as follows.

- TestCard.cpp — This will demonstrate a (simple) main application which makes use of a CreditCard class presumed to be defined elsewhere.
- CreditCard.h — This will be a 'header' file which gives the bare-bones definition for the CreditCard class, including the declaration for all member data and the signatures for all member functions.
- CreditCard.cpp — This file will contain the low-level implementations of many of the functions associated with the CreditCard class, include the bodies of many of the methods, and perhaps even some closely associated functions (such as our overload output operator in this case).

Before continuing with the detailed examination of these three files, it will probably help if we briefly overview the issue of compilation in such a setting.

Compilation

With the source code broken across many files, we have to examine the compilation process more carefully. Though it is possible to take all files at once and compile the entire project, it is also possible to do a piecewise compilation of individual component of the project. Those individual components do not form executables, but instead are compiled into an intermediate form known as object code (.o suffix) which are eventually linked together to form a true executable.

The advantage of separately compiling components in a large system is that if you make a change to only one component, you only need to recompile that one component and then relink, rather than having to recompile the entire program from scratch. This can be a great time saver on very large projects.

But to be able to partially compile one particular component of a larger system, there needs to be some knowledge about those other components, even if full knowledge is not needed. That is, if one component is relying upon a class which is not being compiled as part of that component, there must be at least enough information about that outside piece of code to check the ***syntax*** of the component being compiled. For example, the members of a class must be known, including the full ***signature*** of every possible function which can be called. On the other hand, the actual ***body*** of code which will execute when a function is called is not needed when checking the syntax of the calling code.

In the end, to get an eventual executable, all of the various components must be ***linked*** together. This is really a separate stage of the compilation process. When linking the components, the linker must check to make sure that all of the true pieces of code are in place and ready to go. At the same time, it needs to be careful to check for any conflicting information from two or more components. Though definitions may have been used in multiple components, there cannot be multiply defined *function bodies*. For example, an executable must have a `main` routine, which is the one which is run when the program begins. If more than one main routine is found when linking components together, the link fails. For this reason, we do not ever include a main routine as part of our class definitions.

In the end, our compilation can take many forms. For our example, we have two separate potential components, as implemented in `TestCard.cpp` and `CreditCard.cpp` (the file `CreditCard.h` is not itself a component, but instead will be a shared set of definitions included in both components). If we wished to compile just one component, for example, CreditCard, we could do this as

```
g++ -c CreditCard.cpp
```

The `-c` flag instructs the compiler to perform the compilation step but not the link step (the default is to perform both). As a result of this step, assuming the syntax is acceptable, a new file `CreditCard.o` will be created which is the object code for that component. In similar fashion we could compile the TestCard component to build `TestCard.o`. To link those components together to form an executable, we might then execute the following

```
g++ -o TestCard CreditCard.o TestCard.o
```

This takes all of the objects and creates an executable we have named, `TestCard`. Though it was not actually required for us to choose an executable name which matched the source code, it is a common convention. If we did not include the option `-o TestCard` the default behavior is to create an executable named `a.out`. Once we have an executable, we can run it from the command line using the name of the executable as the command itself.

If we did not wish to create the intermediate object codes, we could have performed the entire compilation in one fell swoop as,

```
g++ -o TestCard CreditCard.cpp TestCard.cpp
```

though this may take a bit longer.

Because managing the compilation has become somewhat more complicated when using code from many files, there is a common practice for using something called a ***Makefile*** to manage this process. This allows a user to type the simple command, `make`, to rebuild the entire project. Within the makefile are details as to precisely what files are part of what component. Furthermore, when remaking a project, the system will look to see which files have recently been edited and which have not, and therefore it will not rebuild the entire project from scratch, but only those components which involve a file when has been editied since the last time the project was built.

Makefiles are a great convenience, yet we do not wish to go into the details of how to write your own makefiles. For this reason, **we will provide you with a Makefile for each assignment in this course, to ease this transition.** You do not need to understand how that file works, only that you type `make` to build a new executable.

Breakdown of code across files

With a better understanding of the compilation/linking process, we now wish to discuss the more immediate concern of a programmer, which is how code is broken across files and what needs to be done differently in this regard.

As discussed, we have broken our original CreditCard class definitions across two files, with suffixes .h and .cpp. The header file (.h) serves as the minimal information which is

needed for another program to be able to use the CreditCard class. So for example, within the TestCard.cpp file, the first few lines are as follows

```
#include "CreditCard.h"    // provides the CreditCard class
#include <iostream>        // provides input/output classes
#include <string>          // provides string
```

The commands starting with the # character, are preprocessor directives and are used in an early stage of the compilation process. The first line has the effect of including all definitions from the given file, `CreditCard.h`. In essence this directive behaves exactly as if it were replaced literally by the entire contents of the given file. The second and third lines are similar directives; the reason for the different syntax in specifying the file is that our CreditCard.h file was presumed to be sitting in the same directory, whereas `iostream` and `string` are presumed to be among the standard libraries which exist system-wide.

Whereas the header (.h) file serves as a definition which can then be included from within other files when needed, the .cpp file can be used to pull aside some of the actual low-level details such as the function implementations. This serves two purposes. Conceptually, someone writing code to use a class need only know about the interface for using that class. The actualy implementation details are not relevant to that outside code. By placing those implementations in another file, this streamlines the header file and makes it more readible as a direct form of documentation for how to use a class. Therefore, this separation serves to aide other programmers.

More significantly, the separation may be necessary for the compilation process. Recall that for individual components to be compiled, their syntax must be checked. The header file provides all of the significant details for using a particular piece of code, including the definition of a class, and the signatures of all functions. By sharing the same header file, multiple components can ensure that they are relying precisely upon the same interface.

At the same time, we must be very careful because these header files may get included from many different places in a larger project, sometimes from different components and sometimes even more than once in the same component. For this reason, when defining a header file which is expected to be used in a larger context the following technique is standard. The very first commands in our `CreditCard.h` header file are the following

```
#ifndef CREDIT_CARD_H
#define CREDIT_CARD_H
```

And the very last line of the file is

```
#endif
```

Everything else falls between these two pieces. As these command start with the # symbol, they are in fact commands relevant to the preprocessing stage of the compilation. They provide a conditional structure, not in the context of a C++ conditional, but a conditional for the compilation process. Intuitively what this combination of lines does is to prevent the header from being read twice within the same compilation cycle. We might read this to say "If you have not already read this file, go right ahead (but otherwise skip it)." The way

we know whether or not is by defining a compiler variable, in this case `CREDIT_CARD_H`. If that variable had not yet been defined, then presumably this file should be read. By then defining this variable on the second line above, this presumably ensures that the next time this very file is included, the compiler will recognize that it has already seen it.

The final issue to address is the `CreditCard.cpp` file. As we mention before, we have broken the original class definition into two files, the header and this implementation file. What we have done is to move many (though not all) of the function *bodies* into this implementation file. Whereas our previous version defined the chargeIt method *within the original class definition*, as follows

```
bool chargeIt(double price) {    // make a charge
  if (price + balance > limit)
    return false;                // over limit
  else {
    balance += price;
    return true;                 // the charge goes through
  }
}
```

our latest version of the project has broken this into two pieces. Within the class definition of file `CreditCard.h` is the declaration of the signature of this method.

```
bool chargeIt(double price);
```

This looks just like the original, except there is no body given, and instead just a semicolon. Remember, for someone to use this class (or a compiler checking syntax), this signature provides all of the necessary information about its use. The body is only meaningful when it comes time to execute the program.

The body of this method has been relocated to within the `CreditCard.cpp` file, where it appears as follows.

```
bool CreditCard::chargeIt(double price) {    // make a charge
  if (price + balance > limit)
    return false;                // over limit
  else {
    balance += price;
    return true;                 // the charge goes through
  }
}
```

Notice that this is almost identical to the original, but there is one significant change. If you look carefully at the first line, you will see the insertion of the characters `CreditCard::` immediately prior to the `chargeIt` identifier. This is called a ***scope resolution***. In our self-contained version, it was quite clear that the chargeIt method was being defined within the scope of the CreditCard class. This was clear based upon the nesting of curly braces in that implementation. Yet now that we have accepted that different pieces of a larger

project can be placed within different files, we must be more careful to clarify our intentions. Though we have named our file `CreditCard.cpp` to suggest its purpose, the filenames are actually irrelevant to the compiler. The code in this file is no longer explicitly contained with the formal class definition. Therefore we must make explict that what we are defining here is the body of code associated with the `CreditCard::chargeIt` method (as opposed, perhaps to an `Atom::chargeIt` method of a hypothetical Atom class elsewhere in the larger context).

At this point, there are a few more issues that we may have swept under the carpet, but we choose to consider this the end of the "introduction."