C++ Concrete Classes

Intermediate C++
18-26 August 1997
Rob Kennedy
Liz Sexton-Kennedy

☞ **Terminology** [Stroustrup3]: ***Concrete*** versus ***Abstract*** Classes

☞ **Concrete Class: A Small, Efficient Abstraction**
Much as the built-in types in C++ are built for efficiency, concrete classes are meant to emphasize run-time efficiency over OO extensibility. They can be thought of simple user-defined types without concern for all of OO philosophy. A concrete class has no ***virtual*** methods, as in: virtual method() ;

☞ **Abstract Class: An OO Extensible Abstraction**
An abstract class only represents an interface and is a base class for other classes. An instance of an abstract class cannot be created. Abstract classes emphasize OO extensibility over run-time efficiency. An abstract class has at least one ***pure virtual*** method, as in: virtual method() = 0 ;

☞ **Large C++ programs will have both kinds of classes.**

☞ **Some classes will be combinations of both extremes.**

Example: Trybos Bank Name

Intermediate C++
18-26 August 1997
Rob Kennedy
Liz Sexton-Kennedy

☞ **Object Statement: Trybos Bank Name**
A Trybos bank name is a sequence of 4 printable ASCII characters. It is used to identify the structural family to which a Trybos bank belongs. We expect heavy use in comparisons to locate a bank. A bank name must have a logical representation which is independent of machine byte-order.

☞ **Design Statement: This is a concrete class candidate.**

☞ **Basic Concrete Class Form:** [Coplien] "Orthodox Canonical Form"

➨ Default Constructor: Class(void) ;

➨ Copy Constructor: Class(const Class& arg) ;

➨ Assignment: Class& operator =(const Class& arg) ;

➨ Destructor: ~Class(void) ;

➨ RDK: Comparison: bool operator ==(const Class& arg) ; \\ also operator !=()

➨ RDK: Conversion Constructors... if needed.

TRY_Bank_Name Interface...

```
#ifndef    _TRY_Bank_Name_hh_
#define    _TRY_Bank_Name_hh_  ($Id:)
class string ;    \\ Do not need to include entire class definition if only reference to it is used here
class TRY_Bank_Name
{  public:
   TRY_Bank_Name(void) ;
   TRY_Bank_Name(const TRY_Bank_Name& name) ;
   TRY_Bank_Name&    operator =(const TRY_Bank_Name& name) ;
   ~TRY_Bank_Name(void) ;
   friend bool    operator ==(const TRY_Bank_Name& name, const TRY_Bank_Name& name2) ;
   TRY_Bank_Name(const char *c_string) ;                    \\ traditional C string
   TRY_Bank_Name(const string& cpp_string) ;                \\ Note: const reference argument
} ;
#endif    \\ _TRY_Bank_Name_hh_
```

- **Missing: Error-handling**
  Many of our C++ compilers do not handle exceptions yet. I add methods like "is_valid()" to test if object is in "valid" state.   Error: my_name = "3CH" ;
- **Missing: Object state output -** This is usually handled with a friend:
  friend ostream& operator <<(ostream& s, const TRY_Bank_Name& name) ;

And Its Implementation...

```
#ifdef     _TRY_Bank_Name_hh_
#define    _TRY_Bank_Name_hh_  (0.1.0)
#include <string>     \\ Now I use a string method, so I must include entire class definition
class TRY_Bank_Name
{  public:
   TRY_Bank_Name(void)                          {_int_rep = 0 ; }      \\ 0 == invalid state
   TRY_Bank_Name(const TRY_Bank_Name& name)
                                                {_int_rep = name._int_rep ; }
   TRY_Bank_Name&    operator =(const TRY_Bank_Name& name)  \\ also operator !=()
                                                { _int_rep = name._int_rep ;  return(*this) ; }
   ~TRY_Bank_Name(void)                         { }
   bool    operator ==(const TRY_Bank_Name& name1, const TRY_Bank_Name& name2)
                                                { return(name1._int_rep == name2._int_rep) ; }
   TRY_Bank_Name(const char *c_string)          { convert_4chars(c_string) ; }
   TRY_Bank_Name(const string& cpp_string)  { convert_4chars(string.data()) ; }
   bool    is_valid(void)  const                {return(_int_rep != 0) ; }    \\ also method is_invalid()
   friend ostream& operator <<(ostream& s, const TRY_Bank_Name& n) ;  \\ implemented elsewhere
private:
   void convert_4chars(const char* c_string) ;    \\ convert 4 chars to _int_rep; implemented elsewhere
   int    _int_rep ;           \\ integer representation of name, used for efficient comparisons
} ;
#endif    \\ _TRY_Bank_Name_hh_
```

And Its Use.


```
TRY_Bank_Name my_name ;
TRY_Bank_Name your_name("YOUR") ;
TRY_Bank_Name other_name(your_name) ;

if (my_name.is_invalid())   cout << "My name is invalid." << endl ;
if (your_name.is_valid())   cout << "Your name is valid." << endl ;

my_name = other_name ;
if (my_name == your_name)
  { cout << "Now we have same name of " << my_name << endl ;
  }


my_name = "SELF" ;    \\ Note: No such operator in class, yet this still works!
cout << "And now my name is " << my_name << endl ;
```


Semantics in C++ Classes

Intermediate C++
18-26 August 1997
Rob Kennedy
Liz Sexton-Kennedy

- ☞ **Constructors and Assignments**
  For every constructor, there should be a semantically related assignment method. For example, the copy constructor and assignment(own_class). It is not always simple to define one in terms of the other, but they can use a common private implementation method if assignment is non-trivial.
- ☞ **Assignment and Comparison**
  While not usually pointed out by experts, assignment(own_class) implies the ability to test for equality and inequality between two objects of the same class.
- ☞ **Operations and Their Logical Negatives**
  If you provide an operation like operator ==() in a class, you should also provide its logical negative, operator !=(). Always define the negation in terms of the result of the original operation. For the TRY_Bank_Name class:
  friend bool operator !=(name1, name2) { return(! (name1 == name2)) ; }
- ☞ **Other Examples:** pre-increment (++var) and post-increment (var++)
  PS: pre-increment = increment; post-increment = copy then increment

More Refinements

- ***explicit* Qualifier on Constructors -** To prevent a constructor from "implicitly" converting a type to another. Implicit conversion may be useful, but may also hide user misunderstandings, mistakes until program is running.
- ***const* Qualifier -** Always declare an argument to a method to be const if its value will not be changed by the method. Also, declare methods to be const if the logical state of an object is unchanged by them. Example: is_valid()
- **Reference Arguments -** Except for built-in type arguments, always use const references for input arguments whenever possible.
- ***mutable* Qualifier on Data Members -** Some classes differentiate between logical and physical state. This means its data members may change value inside a const method, but the logical state of the object is unchanged. Qualify such a data member as mutable to allow logically const methods.
- **Class Constants -** Some classes require special values to be public. A class constant can be implemented by a ***static*** method which accesses a static data member. That data member is initialized in a compiled source file.

Summary: Concrete Classes

- **Concrete classes represent small, efficient data types.**
- **Start from basic class form to develop a full interface.**
- **Refine behavior advertised by interface with appropriate use of const, const reference arguments, explicit, mutable, class constants, and so on....**
- **Maintain semantic relationships between elements in a class interface. Do not leave out semantic equivalents.**
- **Make the interface comprehensible; comment often.** C++ alone does not completely define all aspects of a class or the abstraction that it implements. Do what you can in C++, then rely on commentary and doc.
- **Inlining:** Methods which are 1 - 2 lines long should be inlined for greatest run-time efficiency; either in the class declaration, after it, or in separate file. Longer methods should be compiled in a separate file (for same reason!).