ABSTRACT CLASSES USING VIRTUAL FUNCTIONS

Abstract Classes

A class that contains at least one pure virtual function is said to be *abstract*. Because an abstract class contains one or more functions for which there is no definition (that is, a pure virtual function), no objects of an abstract class may be created. Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes. Although you cannot create objects of an abstract class, you can create pointers and references to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base-class pointers and references to select the proper virtual function

Using Virtual Functions

One of the central aspects of object-oriented programming is the principle of "one interface, multiple methods." This means that a general class of actions can be defined, the interface to which is constant, with each derivation defining its own specific operations. In concrete C++ terms, a base class can be used to define the nature of the interface to a general class. Each derived class then implements the specific operations as they relate to the type of data used by the derived type.

One of the most powerful and flexible ways to implement the "one interface, multiple methods" approach is to use virtual functions, abstract classes, and run-time polymorphism. Using these features, you create a class hierarchy that moves from general to specific (base to derived). Following this philosophy, you define all common features and interfaces in a base class. In cases where certain actions can be implemented only by the derived class, use a virtual function. In essence, in the base class you create and define everything you can that relates to the general case. The derived class fills in the specific details. Following is a simple

example that illustrates the value of the "one interface, multiple methods" philosophy. A class hierarchy is created that performs conversions from one system of units to another. (For example, liters to gallons.) The base class **convert** declares two variables, **val1** and **val2**, which hold the initial and converted values, respectively. It also defines the functions **getinit( )** and **getconv( )**, which return the initial value and the converted value. These elements of **convert** are fixed and applicable to all derived classes that will inherit **convert**. However, the function that will actually perform the conversion, **compute( )**, is a pure virtual function that must be defined by the classes derived from **convert**. The specific nature of **compute( )** will be determined by what type of conversion is taking place.

```
// Virtual function practical example.
#include <iostream>
using namespace std;
class convert {
protected:
double val1; // initial value
double val2; // converted value
public:
convert(double i) {
val1 = i;
}
double getconv() { return val2; }
double getinit() { return val1; }
virtual void compute() = 0;
};
// Liters to gallons.
class l_to_g : public convert {
public:
l_to_g(double i) : convert(i) { }
void compute() {
val2 = val1 / 3.7854;
}
```

```
};
// Fahrenheit to Celsius
class f_to_c : public convert {
public:
f_to_c(double i) : convert(i) { }
void compute() {
val2 = (val1-32) / 1.8;
}
};
int main()
{
convert *p; // pointer to base class
l_to_g lgob(4);
f_to_c fcob(70);
// use virtual function mechanism to convert
p = &lgob;
cout << p->getinit() << " liters is ";
p->compute();
cout << p->getconv() << " gallons\n"; // l_to_g
p = &fcob;
cout << p->getinit() << " in Fahrenheit is ";
p->compute();
cout << p->getconv() << " Celsius\n"; // f_to_c
return 0;
}
```

The preceding program creates two derived classes from **convert**, called **l_to_g** and **f_to_c**. These classes perform the conversions of liters to gallons and Fahrenheit to Celsius, respectively. Each derived class overrides **compute( )** in its own way to perform the desired conversion. However, even though the actual conversion (that is, method) differs between **l_to_g** and **f_to_c**, the interface remains constant. One of the benefits of derived classes and

virtual functions is that handling a new case is a very easy matter. For example, assuming the preceding program, you can add a conversion from feet to meters by including this class:

```
// Feet to meters
class f_to_m : public convert {
public:
f_to_m(double i) : convert(i) { }
void compute() {
val2 = val1 / 3.28;
}
};
```

An important use of abstract classes and virtual functions is in *class libraries*. You can create a generic, extensible class library that will be used by other programmers. Another programmer will inherit your general class, which defines the interface and all elements common to all classes derived from it, and will add those functions specific to the derived class. By creating class libraries, you are able to create and control the interface of a general class while still letting other programmers adapt it to their specific needs. One final point: The base class **convert** is an example of an abstract class. The virtual function **compute( )** is not defined within **convert** because no meaningful definition can be provided. The class **convert** simply does not contain sufficient information for **compute( )** to be defined. It is only when **convert** is inherited by a derived class that a complete type is created.