The object is integration

How it works in the C++ language

Andy Register

The new thing in the computer programming arena is object-oriented programming (OOP). To review (the previous article), OOP distinguishes itself from conventional programming in three important ways: encapsulation, inheritance and polymorphism.

Encapsulation describes the organization of both code and data into powerful units called objects. This organization, a natural extension of a data-structure, creates an implicit relationship between object-code and object-data. Inheritance describes the organization of objects into an object hierarchy. A *child-object* is said to inherit and specialize the behavior of one or more *parent-objects*. Polymorphism describes the ability of an object to react to a command with a reaction consistent with the object.

Of course, there are many methods of effecting encapsulation, inheritance, and polymorphism. Several object-oriented programming languages (OOPLs) have been developed including Smalltalk, Turbo Pascal with Objects, and C++ (pronounced C plus plus).

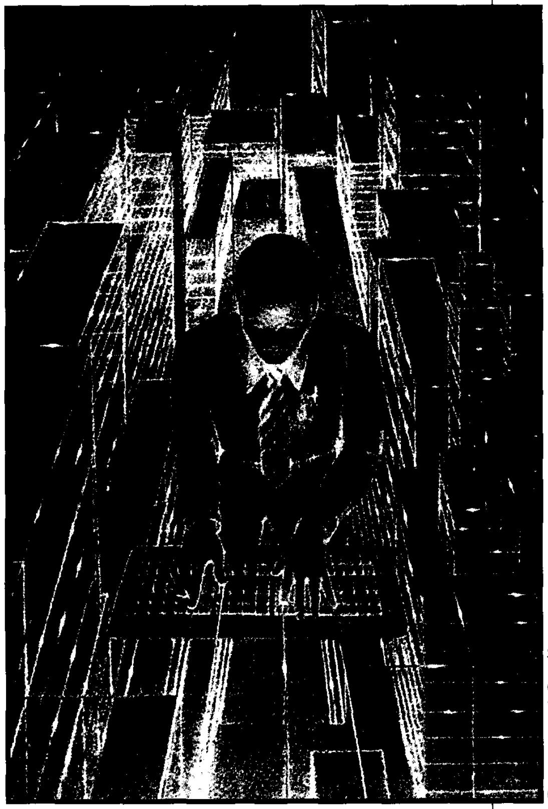
C++ is a popular OOPL among electrical engineers. This popularity is due in part to a strong similarity to ANSI-C. The similarity allows an engineer to first use C++ as a "better C" by learning encapsulation techniques. This first step, encapsulation using C++, is the topic of this article. Using C++ as a "better C," however, should not be the ultimate goal. The full power of C++ can only be realized by mastering all the underlying OOP mechanisms.

Objects

A properly designed object functions as a "black box." The interface is accessible and well defined while the inner-workings are guarded and possibly unknown.

Objects in C++ can be identified with the keyword class, struct or union. Although all three are similar, standard programming convention reserves the new keyword class to denote an object definition. (Struct and union retain their traditional ANSI-C characteristics.) A class allows both data and the functions that operate on the data to be defined within the same structure. The data are called *member variables.* The functions are called *member functions.*

To achieve "black box" operation, the member variables and member functions are defined with one of three access levels: public, protected, and private. Public members play the role of the "black box" interface and private members play the role of the inner-workings. Protected access is used in an inheritance hierarchy to allow seamless access to certain inherited behaviors.



The Image Bank/Gary Kaemmer

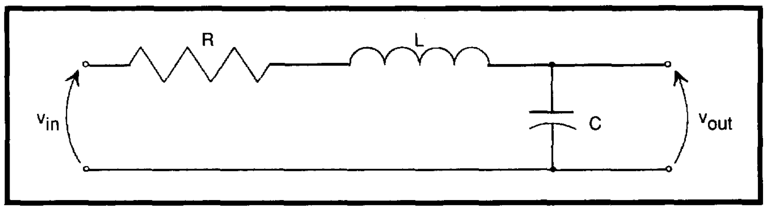


Fig. 1 Low-Pass RLC filter.

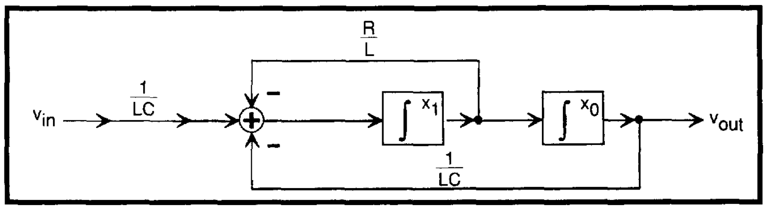


Fig. 2 Block diagram representation of the RLC filter circuit.

In a properly designed object, the member variables are private and public member functions are used to manipulate these private variables. The advantages of this organization are many. Code changes are localized within an object. An object may be isolated and tested before it is included in a larger program. Program side-effects are minimized because private member variables are manipulated only through the public interface. These advantages can pay large dividends to the reliability and maintainability of programs.

With a little effort, it is possible to create a class that is indiscernible from a built-in data type such as *int* or *float*. This type of class is called a concrete data type. Concrete data types can be assigned, declared, and passed as parameters. The convenience of using concrete data types definitely makes the effort worthwhile.

The easiest way to understand the process of creating a C++ object is to create one. For this task, we should create an object that is simple, well understood, and useful. For electrical engineers, a state-variable class meets all the above criteria. State-variables occur in any problem described by a system of differential equations including electrical circuits, rotating machines, and control systems. Solving these problems on a computer requires numerical integration. The state-variable class that is created encapsulates numerical integration as a concrete data type.

Numerical integration

The numerical techniques for integrating a system of differential equations can be segregated into three groups: predictor, predictor-corrector, and single-step. Although generally less accurate, predictor techniques are the easiest to set up because they rely only on previous inputs and/or previous outputs. Predictor techniques are also flexible because they may be used to evaluate the integral of a sampled-data signal or a function whose solution is expensive to generate.

If the predictor technique relies solely on a history of previous inputs, it is called an Adams-Bashforth (AB) formula. The AB formulas are represented by the nth-degree polynomial,

$$x_{i+1} = x_i + \sum_{k=0}^{n-1} a_{-k}\dot{x}_{i-k}$$

where n represents the order, subscripts denote the sample time-step, and the overdot represents a derivative with respect to time. In the above equation $\dot{x}_{i-k}$ represents a time-history of inputs to the state-variable and the integral, x_{i+1}, consists of a linear combination of this input history.

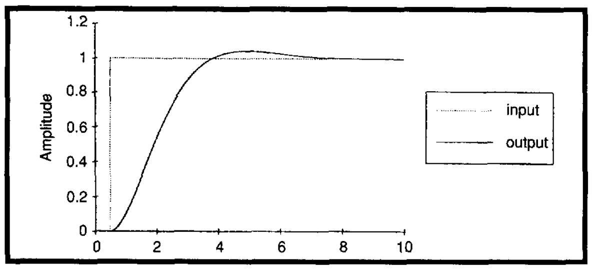


Fig. 3 Plot of the RLC filter input and output amplitude versus time.

The first and second order AB formulas are the familiar Euler and trapezoidal integration methods, respectively. AB coefficients are well known and can be found in any text on numerical integration. The fourth-order AB polynomial will be used in C++ state-variable class example. The equation is,

$$x_{i+1} = x_I +$$

$$\frac{\Delta t}{24}\left(55\,\dot{x}_{\mathrm{i}} - 59\dot{x}_{\mathrm{i-1}} + 37\dot{x}_{\mathrm{i-2}} - 9\dot{x}_{\mathrm{i-3}}\right)$$

Defining the object

AB techniques require three pieces of information: the integration order, the current value of the integral, and an input history. An integer variable can be used to hold the integration order. A floating point variable can be used to hold the current value, and an array of floating point values can be used to hold the input history. Even though the order is fixed, setting the order and consequently setting the size of the input history array will be deferred to runtime. In other words, memory for the input history array will be dynamically allocated. In C++ syntax, the data-structure requires at least the following member variables:

```
int order;
double integral;
double *in_history;
```

Object lifetime

The member functions used to access member variables have different purposes. A few of the more common classifications are constructor, destructor, inspector, and manipulator. For brevity, the constructor and destructor are often abbreviated *ctor* and *dtor*, respectively. The lifetime of an object begins with the constructor and ends with the destructor. The constructor builds and initializes the object. This includes reserving memory for the member variables and setting initial values. The destructor destroys the object and frees any space the object may have occupied. Every object must have at least one constructor and only one destructor. An inspector is used to return a value without changing the object while a manipulator is used to change some aspect of the object. Inspectors and manipulators form the public interface of the object.

A concrete data type will have member functions that include a default constructor, a copy constructor, an assignment operator, and a destructor. The default constructor is a special constructor that can be called without an argument. This allows simple declarations and the creation of arrays of objects. The copy constructor is used to create a new object that is a copy of an existing object. This allows an object to be passed by value or returned by a function.

An assignment operator is similar to a copy constructor except that nothing is created. The assignment operator allows an object to appear on the left-hand-side of an equal sign.

Finally, the destructor destroys the object when it goes out of scope. If any of these four member functions are omitted from the class definition, the compiler will generate its own version. In all but the most simple cases, the compiler generated member functions will not be adequate.

This is particularly true when the object contains a pointer. Special precautions must be taken. The compiler generated copy constructor and assignment operator perform what is known as a *shallow copy*. A shallow copy will copy the address of a pointer but not the underlying element (or elements if the pointer is the beginning of an array). This can cause two objects to unintentionally share the same data. If one object goes out of scope, the shared data may be destroyed, leaving the other object to reference arbitrary data.

By explicitly defining a copy constructor, a *deep copy* can be performed. A deep copy creates a new pointer and copies the values into the new pointer location. The new object now has its own copy of the data.

By definition, constructors take the same name as the class. Indeed, all constructors will appear to have the same name. In C++, two or more functions can have the same name as long as the argument lists are different. The compiler can differentiate between functions based on the number and type of arguments. Argument based function selection is valid for all C++ functions not just constructors or member functions. This feature helps simplify syntax and enhances the maintainability of C++ programs.

Destructor names are denoted by appending a tilde, "~," to the front of the class name. A destructor takes no arguments and has no return type. Since the job of the destructor is to reverse the work of the constructor, dynamic allocation by the constructor must be balanced with deallocation by the destructor.

Syntactic sugar

An assignment operator is one example of a C++ feature called *operator overloading*. Operator overloading is a bit of "syntactic sugar" that allows programs to be written using operator syntax instead of procedural syntax. To over-simplify the point, imagine that y, a, and b are matrices. Languages without operator overloading represent matrix addition using a syntax comparable to,

```
add (y, a, b).
```

With operator overloading, matrix addition is represented in a very clear syntax given by,

```
y = a+b.
```

Thus, when properly used, operator overloading enhances the readability and consequently the maintainability of programs.

Each class has the opportunity to overload and redefine the definition of any overloadable operator. The list of overloadable operators is large, including all the expected operators: assignment (=), mathematical (+,-,*,/), logical (&,|), and some unexpected operators: type-conversion, "[]," "()," "->." The "operator" keyword along with the operator symbol is used to define the overload.

Following the above discussion, the member functions required to make the state-variable class a concrete data type can be written in C++ syntax as,

```
    SV(void);
// default ctor
    SV(const SV &sv);
// copy ctor
    SV& operator=(const SV &sv);
// assignment operator
    ~SV(void);
// dtor
```

To an ANSI-C programmer, there are several new items in the above definitions. The double slash (e.g., //) is a one line comment delimiter. The comment begins with the double slash and ends at the end of the line. The ANSI-C comment notation (e.g., /* */) is still recognized.

The argument syntax, (*const* SV *&sv*), is also new. The *const* keyword in concert with the "&" symbol emulates a fast, pass-by-value function argument. In reality, the "&" specifies a reference or pointer to the object and *const* constrains any access as read-only. Thus, speed is obtained by passing a pointer and pass-by-value emulation is obtained using read-only access. For objects, pass-by-const-reference is always preferred to pass-by-value.

Custom behavior

The inspectors and manipulators included in the public interface define the behavior of the class. In the state-variable class, four operations must be performed: input the current value, calculate the integral, update the history, and view the result. The input function will be accomplished by overloading the leftshift operator, <<. Viewing the result will be accomplished using a type conversion to *double*. The descriptive member functions *Integrate* and *UpdateHistory* will also be used. *Integrate* will calculate the integral, call *UpdateHistory* and return the integral. Since *Integrate* will perform the history update, a user should never be allowed to independently update the history. Hiding *UpdateHistory* with private access will prevent a user from calling the function. This type of hidden private function is often called a helper function.

Object module strategy

```
/* file sv.h */
#if !defined(SV_H)
#define SV_H
#include <iostream.h>
#include <stdlib.h>

class SV{
    static double Step;
    int order;
    double integral;
    double *history;
    void UpdateHistory(void);
public:
    SV(int Order=4);
    SV(const SV &sv);
    SV& operator=(const SV &sv);
    ~SV(void) { delete[] history; };
    void operator<<(double inp) { history[0] = inp; }
    operator double(void) const { return integral; }
    double Integrate(void);
} ;

inline SV::SV(int Order) {
    if (Order != 4) {
    cerr << "SV Error: Order != 4 not defined." << endl;
    exit(1);
    }
    order = Order;
    history = new double[order];
    integral = 0.0;
    for (int i = 0 ; i < order ; i++)
    history[i] = 0.0;
}

inline SV::SV(const SV &sv) {
    order = sv.order;
    history = new double[order];
    integral = sv.integral;
    for (int i = 0 ; i < order ; i++)
    history[i] = sv.history[i];
}

inline double SV::Integrate(void) {
    // fourth order Adams-Bashforth method
    integral += (1.0/24.0)*Step*
( 55.0*history [0]
- 59.0*history [1]
+ 37.0*history [2]
- 9.0*history [3]);
    UpdateHistory ();
    return integral;
}

inline void SV::UpdateHistory(void) {
    for (int i = (order-l) ; i > 0 ; i—)
    history[i] = history[i-1]; }

inline SV& SV::operator=(const SV &sv) {
    if (this == &sv)
    return *this;
    if (order != sv.order) {
    cerr << "SV Error: Cannot equate" << endl;
    exit(1);
    }
    integral = sv.integral;
    for (int i = 0 ; i < order ; i++)
    history[i] = sv.history[i]
    return *this
}
#endif // SV_H
```

Butterworth Code

```
/* file butter2.cpp - second order Butterworth filter */
#include <math.h>
#include "sv.h"

// set the state-variable time step
const double timestep = 0.01;
double SV::Step = TimeStep;

// Butterworth normalized to 1 rad/sec
const double R = sqrt(2.0);
const double L = 1.0;
const double C = 1.0;

void main() {
   SV x0, x1;          // the states
   double vin, vout;  // input and output
   cout << "time\t" << "input\t"~ << "output" << endl;
   for (double t=0.0 ; t<10.0 ; t = t+TimeStep) {
   // create a step input
   if (t >= 0.5) vin = 1.0; else vin = 0.0;
   // find the state-variable inputs
   x0 << xl;
   xl << ( (1.0/(L*C))*(vin - x0) - (R/L)*x1 );
   // perform the integration
   x0.Integrate();
   x1.Integrate();
   // find the filter output
   vout = x0;
   // print the output
   cout << t << "\t" << vin << ~"\t" << vout << endl;
   }
}
```

The state-variable class

The design of the state-variable class is now complete and the resulting code is included below. In a programming environment, the code can be directly inserted, stored as an include file or broken-up into an include file and a separately compiled object module. For large projects, the object module strategy will reduce compile time. (See box on page 39.)

Glancing through the listing confirms that the syntax is very close to ANSI-C. Most of the new C++ features are easy to understand. However, there may be several new constructs that are unfamiliar. These constructs are important enough to mention briefly.

In the include directive, there is a file named *iostream.h.* To over-simplify, this file implements a new I/O mechanism by overloading the "<<" and ">>" operators for output and input respectively. This new I/O mechanism is called *stream I/O* and is an enhanced, C++ replacement for the ANSI-C printf() function. The stream mechanisms are also overloadable. Unfortunately, beyond this, streams are well outside the scope of this introduction.

The operators, *new* and *delete*, are used to allocate and deallocate memory. Although the ANSI-C functions *malloc* and *free* could be used, *new* and *delete* allow more flexibility. One major enhancement is that *new* and *delete* activate the class constructor or destructor. For C++ programs, *new* and *delete* should be used exclusively.

The use of the *static* keyword for a member variable is another addition. Declaring the member variable *static* allows the same variable to be shared across every state-variable in a program. Here, the constant time-step required by the AB algorithm is shared by all state-variables. This restriction helps keep the system in synchronization.

Finally, the *inline* keyword is a hint to the compiler that certain functions may be expanded to run without a function call. This has performance advantages for small procedures where the calling overhead may be a significant percentage. The disadvantage is that the code size may become large if all functions are defined *inline*.

Critically damped

Now that the class has been designed and coded, an example of its use is appropriate. To explore the use of the state-variable class and confirm its operation, a low-pass RLC filter circuit will be examined. Due to space considerations, however, the example will be brief. The circuit diagram is shown in Fig. 1 and the corresponding transfer function is,

$$\frac{v_{out}(s)}{v_{in}(s)} = \frac{\frac{1}{LC}}{s^2 + \frac{R}{L}s + \frac{1}{LC}}$$

The circuit can also be represented by the block-diagram shown in Fig. 2. The set of differential equations taken from Fig. 2 is,

$$\dot{x}_0 = x_1$$
$$\dot{x}_1 = \frac{1}{LC}v_{in} - \frac{1}{LC}x_0 - \frac{R}{L}x_1$$
$$v_{out} = x_0$$

The differential equations can be directly implemented using the state-variable class. If R, L, and C are properly chosen, a Butterworth response can be achieved. The code to implement the Butterworth RLC filter is shown in box.

Notice how the type-conversion to double (e.g., operator double in the SV class definition) allows state-variables to be included in mathematical expressions. A plot of the output is shown in Fig. 3. The output amplitude is comparable to an ideal Butterworth response. The error is insignificant and acceptable for many applications.

Conclusion

A first cut at a useful C++ class has been accomplished. Building the class structure allowed many object-oriented ideas to be investigated and resulted in a concrete data type that is useful in many applications. An RLC filter example helped to show how to use the class. Although usable, the state-variable class could be improved.

There are some useful methods that were not included. An inspector for the step-size, and a mechanism to automate the time loop are examples of possible enhancements. More powerful integration techniques could also be investigated. Using additional techniques might be a good place to take advantage of inheritance to build a "toolbox" of several integration methods. By building on this introduction, you can improve the state-variable class and build additional classes for other needs.

Read more about it

- Grady Booch, "Object Oriented Design with Applications," Benjamin/Cummings, 1991.
- Bjarne Stroustrup, "The C++ Programming Language, 2nd ed.," Addison-Wesley, 1991.
- William Press, et. al., "Numerical Recipes in C," Cambridge University Press, 1988.
- UseNet News, comp.lang.c++.

Acknowledgments

The investigation leading to this paper was funded in part by the NSF/STA 1992 Japan Summer Institute Program. The author thanks the researchers in the Mechanism Division of the Mechanical Engineering Laboratory, Tsukuba Science City, Japan, for their help and support.

About the author

Andy Register is a Student member of IEEE and a Ph.D. candidate at The Georgia Institute of Technology. His research interests include both rigid-link and flexible-link manipulator dynamics and control. This includes object-oriented simulation and the investigation of learning control methods.