C/C++ Users Journal™
Advanced Solutions for C/C++ Programmers

June 2001
Vol. 19, No. 6

ALGORITHMS

10 Three Guidelines for Effective Iterator Usage

Scott Meyers

It is easy to write code that uses STL iterators, but beware lest you write code that's brittle or won't port. Here are a few things you might need to know.

22 Implementing Reusable Mathematical Procedures Using C++

Qiang Liu

Algorithms that must call user-supplied functions are typically difficult to reuse. The key to reuse is finding a way to encapsulate the user-defined code.

BOOK REVIEW

30 *Accelerated C++*

reviewed by Francis Glassborow

Maybe you've been hearing the buzz about some "new approach" to learning C++. Here's the book that's got everyone talking.

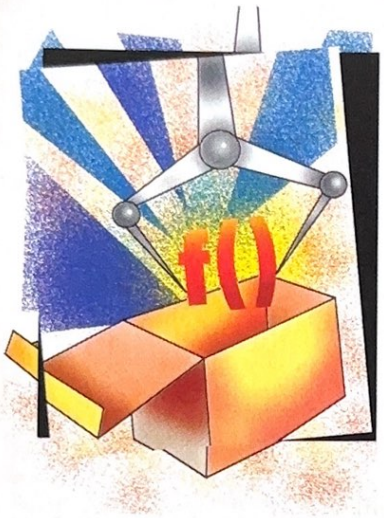
DEPARTMENTS

Editor's Forum – 6
Call for Papers – 64
Advertiser Index – 69
New Products – 71
We Have Mail – 75
Programmer's Market – 76

■ **Visit the *C/C++ Users Journal* website: <http://www.cuj.com>** ■

Cover by Twyla Watson Bogaard.

C/C++ Users Journal (ISSN 1075-2838) is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco, CA 94107 (785) 841-1631. Periodicals postage paid at San Francisco, CA and additional mailing offices. Canadian Publication Agreement Number 1470736. **POSTMASTER:** Send address changes to C/C++ USERS JOURNAL, P.O. Box 52582, Boulder, CO 80328-2582. **Subscriptions:** Annual renewable subscriptions to *C/C++ Users Journal* are $34.95 US, $46 Canada and Mexico, $65 overseas. Payments must be made in US dollars. Make checks payable to *C/C++ Users Journal*. **GST (Canada):L #129065819**



Qiang Liu

Implementing Reusable Mathematical Procedures Using C++

Algorithms that must call user-supplied functions are typically difficult to reuse. The key to reuse is finding a way to encapsulate the user-defined code.

Introduction

"Code reuse" is one of the holy grails in software engineering. A primary argument for the adoption of OO (object-oriented) programming is reusability, a fact that is evident from any books on OO design and programming (for example, see [1]). In practice, code reuse using OO languages like C++ is more difficult to achieve than we might hope. In fact, one author has even asserted that, "C++ has done much to impede the reusability of scientific software," owing to the tendency of C++ programmers to create their own container classes [2].

In this article, I show how to create reusable mathematical procedures in C++. My method relies more on Generic Programming concepts than on OO. (For an introduction to Generic Programming, see [3].) To motivate the discussion, I present an example of a widely used estimation algorithm, Newton-Raphson, which must call a user-defined function, and I show how that user-defined function is typically (and unsatisfactorily) packaged. I then evolve a more satisfactory packaging solution based on the use of templates and operator overloading.

The Newton-Raphson Algorithm

In scientific computation and financial engineering, many numerical methods are generic (in theory, at least) and applicable to a wide class of problems. A well-known example is the Newton-Raphson procedure [4], which finds a numerical solution for the equation $f(x) = 0$. Here, the standard mathematical notation $f(x)$ indicates that f is a function of the variable x. A generalized form of this problem is $f(x, a, b, ...) = 0$, where the function f is defined by more than one variable. In this case, Newton-Raphson attempts to find a solution with respect to the variable x while holding the other variables fixed and treating them as parameters. Two such functions in finance describe the price of a bond and the value of a European call option.

The price of a bond is determined by its yield, coupon, frequency of coupon payments, and maturity; the value of a European call option is a function of its underlying stock price, strike price, interest rate, time to maturity, and volatility [5]. In the first case, a typical problem is to calculate the yield, given the price of the bond and holding the other parameters fixed. In the second case, the volatility is the parameter to be determined given the price of an option traded on the market. This unknown parameter is known as the implied volatility, and it is computed and used widely. In both cases, the algorithm used to calculate these parameters is Newton-Raphson.

Traditionally, the Newton-Raphson method is coded and embedded directly in client applications, because this procedure needs the exact definition of the function for which a solution is sought. As a result, its implementation, with slight modifications for different functions, is repeated over and over again in many places.

The details of the Newton-Raphson procedure, as well as those of many other mathematical routines, should not concern its users. Further, code repetition should be avoided whenever possible. It is only natural to seek a way of implementing such routines as library functions so that their client applications can call them directly. But making Newton-Raphson into a library function must involve packaging the user-defined function (which Newton-Raphson calls) into a form that can be passed as a parameter. The following section describes a common — and problematic — way of packaging the user-defined function.

The Common Approach — Pointer to Function

The task is to make Newton-Raphson into a library routine that finds a solution x for all equations with the form $f(x, a, b, ...) = 0$. The trick is that the implementation of the routine must use (or call) a generic function of the form $f(x, a, b, ...)$, whose exact definition will be provided later by the library's user and made available to the library only at run time. To a C or C++ programmer, one obvious possibility is to pass a pointer to function as an argument to the library routine:

Qiang Liu is a senior programmer/analyst with Highbridge Capital Management in New York City. He holds a Ph.D. in Quantum chemistry from Cornell University. He enjoys programming with C++, VB, and Unix shell scripts. He can be reached at qiangl@hcmny.com.

```
typedef double (*P2F)(double);

double NewtonRaphson(P2F func_of_x, double x_init,
   <some_other_inputs>) {
   ...
   // call function through pointer to function
   double y = func_of_x( x_init );
   ...
}
```

This library routine works fine, but only for functions defined with exactly one argument. In C++, the programmer could define one overloaded library function for each new user-defined function taking an additional argument, but would end up repeating much of the library code — and worse, would not be able to anticipate h many library functions would need to be written.

Another idea is to use optional arguments, as in the following:

```
typedef double (*P2F)(double, ...);
```

This could have been the end of the story. Fortunately and unfort nately, C++ does not allow optional arguments in the way that intended here. A pointer to function has to know the exact numbe and types of arguments to a function, so the signature in this typed will match only functions taking a `double` followed by C-styl `varargs`, not functions taking further specific parameter types.

There might be other ways to pass functions of multiple argument such as trying a function wrapper, but it is n clear to the author how that can be done with out resorting to global variables.

To put it simply, some construction i needed that holds a few parameters used t define a complex function and provides way for a library routine to access this ver function using just one argument. Tha would be an *object* — pure and simple Therefore, I will define a class for the gener al function, f(x, a, b, ...), and call it `FuncObj` (For simplicity, the number of argument will be fixed to three from now on.)

```
class FuncObj {
private:
   double _a;
   int _b;
public:
   FuncObj(double a_in, int b_in);

   // the body that defines the function
   // in terms of x, a, and b
   double theFunc(double x_init);
};
```

You might be tempted to use the library routine defined above by passing a pointer to the member function `theFunc` of the class `FuncObj`. This will not work, however, for two reasons. First, a pointer to member function, which contains the class name as part of its signature, cannot be used in place of a pointer to (a normal) function. Second, a pointer to member function has to be accessed through an object of that class. I address these two problems in the next section. (Note that defining `theFunc` as static would not really solve the problem: then `theFunc` would not be able to access `FuncObj`'s non-static data members, which are needed to hold the other "constant" variables.)

Using a Pointer to Member Function

As I discussed in the previous section, it is necessary to modify the library interface

to use it with the pointer to member function approach. Furthermore, the interface should obviously be defined as a function template so that it will not be locked to one particular class:

```
template <class T>
double NewtonRaphson(T & func, double (T::*func_of_x)(double),
   double x_init, <some_other_inputs>) {
      ...
      // call member function through an object (reference) and
      //   a pointer to member function
      double y = (func.*func_of_x)( x_init );
      ...
}
```

This works, but the syntax is a little bit difficult to read.

One way to make the code simpler and more readable would be to create a typedef for the pointer to member function, just as one was created for the simple pointer to function earlier. In other words, it would be nice to create a typedef with a parameterized type, as in the following:

```
template<class T>
typedef double (T::*P2MF)(double);
```

If the above code were legal C++, then P2MF would be of type "pointer to member function of class T taking a double as a parameter and returning a double." However, C++ does not support template typedefs.

As always, the ultimate solution in computer science is to introduce another level of indirection, in this case, by defining an enclosing template class just to make the typedef legal:

```
template<class T> struct P2MFHelper {
   typedef double (T::*P2MF)(double);
};
```

This constitutes a strange but interesting use of template and typedef. Now I can redefine the library function as:

```
template <class T>
double NewtonRaphson(T & func,
   P2MFHelper<T>::P2MF func_of_x,
   double x_init, <some_other_inputs>) {
   ...
   double y =
      (func.*func_of_x)( x_init );
   ...
}
```

Note that the parentheses around func.*func_of_x are necessary for the code to compile and work correctly, because the function call operator that follows has a higher precedence than the operator.*. Also, note that the helper class and the library function could be combined into one single template class, which would achieve the same goal. However, it doesn't seem like a good design to make the Newton-Raphson procedure a class; needless to say, it would also result in a slightly more complex syntax. This library will be suitable for user-defined functions with various numbers of arguments, provided we define one FuncObj-style class to package each such function.

This approach has two minor problems, however. First, you can no longer pass a pointer to (a normal) function to the NewtonRaphson procedure. Furthermore, it is not elegant and economical to pass both an object and a pointer to member function (although in principle this might be solved by using member function adapters). Below I show how to eliminate these two "nuisances."

Using a Function Object

A function object is a class that has an overloaded () operator, which is the function call operator [6]. Therefore, it is possible to use an overloaded operator () of a function object in place of a function call operator. Below I redefine `FuncObj`, the complex function of many variables, as a bona fide function object:

```
class FuncObj {
public:
   // overloaded operator
   double operator() (double x_init) {
      // the body of the operator
      // implements the function and
      // is defined in terms of x, a,
      // and b
   }
};
```

Now the library routine can be defined more simply, using the function object directly:

```
template <class T>
double NewtonRaphson(T & func,
   double x_init,
   <some_other_inputs>) {
   ...
   // call a function or the operator()
   // of a function object
   double y = func( x_init );
   ...
}
```

Note that only one parameterized argument is passed to the `NewtonRaphson` library routine. This type can be substituted with an object reference or a pointer to a normal function. Furthermore, the syntax of the "function call" inside the library function body is much simpler. As an example, consider how this generic library function can be used to solve numerically the complex equation, $X^3 + 2e^X + 7 = 0$, which does not have an analytic solution. The function object is defined as:

```
class FuncObj {
private:
   double _a;
   double _b;

public:
   FuncObj(double a_in, double b_in) :
      _a(a_in), _b(b_in) {};

   // overloaded operator
   double operator() (double x_in) {
      return ( x_in*x_in*x_in +
         _a*exp(x_in) + _b );
   }
   double solve(double x_in) {
      // use the generic library
      // function
      return NewtonRaphson(*this, x_in,
         other_arguments);
   }
};
```

The library function is used simply in the `main` program as follows:

```
void main() {
   FuncObj fo(2.0, 7.0);

   // call the (Newton-Raphson) library
   // function indirectly
   double solution_1 = fo.solve(-4.0);

   // call the library function directly
   double solution_2 =
      NewtonRaphson(fo, -4.0,
      other_arguments);
}
```

Note that this version of the library function works both inside and outside the function object. `main` actually shows one direct call to the library function by passing an object reference. Furthermore, it is also fine to pass a suitable pointer to function to the library function.

For curious readers, two sample files, `NewtonRaphson.h` and `main.cpp`, are shown as Listings 1 and 2 respectively. The former is a simplified version of the library implementation of the Newton-Raphson procedure, while the latter demonstrates the use of this function in three different ways.

Conclusion

By taking advantage of two powerful C++ mechanisms, I have arrived at a reusable library implementation of the Newton-Raphson method. This library is generic and applicable to many similar problems, provided the client defines an overloaded operator () for that complex function (object) that the library function will call.

The approach described in this article is applicable to many well-known scientific algorithms that solve a class of mathematical problems. Numerical integration for an arbitrary complex function, for example, can be treated in the same fashion. Two key features of the C++ language, namely function templates and operator overloading, make this simple, elegant solution possible. The overhead of making a function an object is negligible. In fact, many real-world applications have already defined complex functions as

classes, to which an overloaded ( ) operator can be added whenever some generic numerical procedure, such as the Newton-Raphson algorithm, is called upon. Therefore, it is possible now to implement a whole suite of mathematical algorithms into local reusable libraries. ❑

References

[1] Jesse Liberty. *Beginning Objected-Oriented Analysis and Design with C++* (Wrox Press, 1998).

[2] Geoffrey Furnish. "Disambiguated Glommable Expression Templates Reintroduced," *C++ Report*, May 2000.

Listing 1: *NewtonRaphson.h — A simplified version*

```
// Newton-Raphson numeric algorithm: find a solution for f( x ) = 0
// Note: func_x maybe a function object or a pointer to function

#ifndef NEWTONRAPHSON_H
#define NEWTONRAPHSON_H

#include <cmath>

template<class T> double NewtonRaphson(
    T & func_x, double x0, double step_x,
    double x_abs_err, int max_iter) {

    enum {MAX_ITER_EXCEEDED, DIVISION_BY_ZERO, X_IN_TOLERANCE}
       NRResult;

    double y0 = func_x(x0);
    double x1 = x0 + step_x;
    double y1 = func_x(x1);

    double x_jump;
    NRResult = MAX_ITER_EXCEEDED;
    for(int i = 0; i < max_iter; ++i) {
        double x1_keep = x1;
        double y1_keep = y1;

        if ( y1 != y0 ) {
            x_jump = y0 * (x1 - x0) / (y1 - y0);
            x1 = x0 - x_jump;
        }
        else {
            x_jump = x1 - x0;
            NRResult = DIVISION_BY_ZERO;
        }

        double err_x = fabs( x_jump );
        if ( err_x < x_abs_err ) NRResult = X_IN_TOLERANCE;

        if ( NRResult != MAX_ITER_EXCEEDED ) break;

        y1 = func_x(x1);
        x0 = x1_keep;
        y0 = y1_keep;
    }

    double nr_value;
    switch ( NRResult ) {
        case X_IN_TOLERANCE:
            nr_value = x1;
            break;
        default:
            nr_value = 0.0;
    }
    return nr_value;
}
#endif  // NEWTONRAPHSON_H
```

— End of Listing —

[3] Matt Austern. *Generic Programming and the STL* (Addison-Wesley, 1999).

[4] John H. Mathews. *Numerical Methods for Mathematics, Science and Engineering*, 2nd Ed. (Prentice Hall, 1992).

[5] John C. Hull. *Options, Futures, and Other Derivatives*, 3rd Ed. (Prentice Hall, 1997).

[6] Bjarne Stroustrup. *The C++ Programming Language*, Special Ed. (Addison-Wesley, 2000).

Listing 2: *Packages a user-defined function as pointer to function and a function object; then calls the function two different ways as a function object and once via pointer to function*

```
// main.cpp

#include <iostream>
#include <cmath>
#include "NewtonRaphson.h"

// (X^3 + a*exp(X) + b) defined as a function:
// one solution is -1.9387
double func(double x_in)
{
    const double A = 2.0;
    const double B = 7.0;

    return x_in * x_in * x_in + A * exp(x_in) + B;
}

// the same function defined as a function object
class FuncObj
{
private:
    double _a;
    double _b;
public:
    FuncObj(double a_in, double b_in)
        : _a(a_in), _b(b_in) {};

    double operator() (double x_in)
        {
            return  x_in * x_in * x_in + _a * exp(x_in) + _b;
        }

    double solve(double x_in)
        {
            return NewtonRaphson(*this, x_in, 0.2, 0.0001, 20);
        }
};

void main() {  // use the Newton-Raphson library in three ways
    FuncObj fo(2.0, 7.0);

    cout << "Call Newton-Raphson indirectly: x = "
         << fo.solve(-10.0) << endl;
    cout << "Call Newton-Raphson directly via FuncObj: x = "
         << NewtonRaphson(fo, -10.0, 0.2, 0.0001, 20) << endl;

    cout << "Call Newton-Raphson directly via Func Ptr: x = "
         << NewtonRaphson(func, -10.0, 0.2, 0.0001, 20) << endl;
}
```

— End of Listing —

CMP

Scott Meyers on Iterators: Three Important Guidelines

JUNE 2001

Visit us on the Web:
www.cuj.com

C/C++ Users Journal

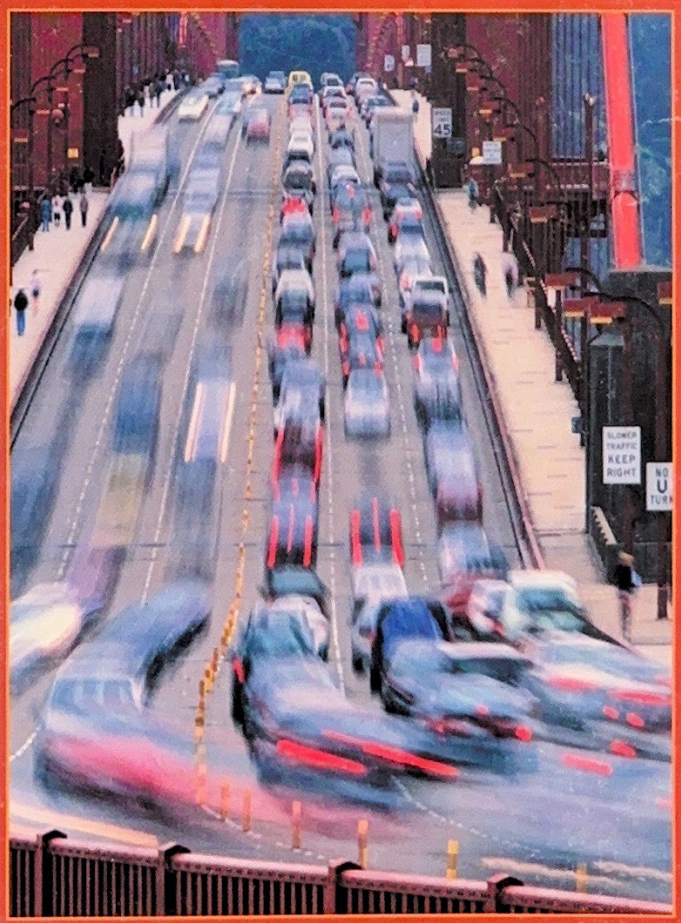
C/C++ Users Journal™

Advanced Solutions for C/C++ Programmers

ALGORITHMS

Reusing Stubborn Algorithms

Packaging Is the Key



Book Review: Accelerated C++

A New Way to Learn C++

Columns by:
Bobby Schmidt
Andrew Koenig
Barbara E. Moo
Steve Dewhurst
Thomas Becker
Randy Meyers
Stan Kelly-Bootle

Vol. 19, No. 6