

Using C++ to Write Automation Controller Software

Russell T. Berman^a
Velocity11, Menlo Park, CA*

Keywords:

C++,
automation,
controller,
scheduler,
programming

The object-oriented programming language C++ is extremely well suited for writing automation control software. Because it is object-oriented, C++ provides elegant mechanisms for developing reusable interfaces. For example, programmers can leverage off-the-shelf graphics or communication packages, or they can instead develop their own libraries. Additionally, end users have the possibility of extending the functionality of their control software by developing their own device drivers or Laboratory Information Management System (LIMS) interfaces.

Because C++ provides mechanisms for multithreading, programmers can use it to implement dynamic scheduling algorithms. Event-driven scheduling is often favored above static scheduling because it results in higher instrument utilization and provides better error-handling facilities. (JALA 2007;12:12–6)

INTRODUCTION

As research laboratories become more automated, new problems are arising for laboratory managers. Rarely does a laboratory purchase all of its automation from a single equipment vendor. As a result, managers are forced to spend money training their users on numerous different software packages while purchasing support contracts for each. This

suggests a problem of scalability. In the ideal world, managers could use the same software package to control systems of any size; from single instruments such as pipettors or readers to large robotic systems with up to hundreds of instruments. If such a software package existed, managers would only have to train users on one platform and would be able to source software support from a single vendor.

If automation software is written to be scalable, it must also be flexible. Having a platform that can control systems of any size is far less valuable if the end user cannot control every device type they need to use. Similarly, if the software cannot connect to the customer's Laboratory Information Management System (LIMS) database,¹ it is of limited usefulness. The ideal automation software platform must therefore have an open architecture to provide such connectivity.

Two strong reasons to automate a laboratory are increased throughput and improved robustness. It does not make sense to purchase high-speed automation if the controlling software does not maximize throughput of the system. The ideal automation software, therefore, would make use of redundant devices in the system to increase throughput. For example, let us assume that a plate-reading step is the slowest task in a given method. It would make that if the system operator connected another identical reader into the system, the controller software should be able to use both readers, cutting the total throughput time of the reading step in half. While resource pooling provides a clear throughput advantage, it can also be used to make the system more robust. For example, if one of the two readers were to experience some sort of error, the controlling software should be smart enough to route all samples to the working reader without taking the entire system offline.

^aThe author works for Velocity11, a company that produces automation control software similar to what is described in this manuscript.

*Correspondence: Russell Berman, MS, Velocity11, Research & Development, 3565 Haven Ave, Menlo Park, CA 94025, USA; Phone: +1.650.846.6666; E-mail: rberman@velocity11.com
1535-5535/\$32.00

Copyright © 2007 by The Association for Laboratory Automation
doi:10.1016/j.jala.2006.07.011

Now that one embodiment of an ideal automation control platform has been described let us see how the use of C++ helps achieving this ideal possible.

DISCUSSION

C++: An Object-Oriented Language

Developed in 1983 by Bjarne Stroustrup of Bell Labs, C++ helped propel the concept of object-oriented programming into the mainstream.² The term “object-oriented programming language” is a familiar phrase that has been in use for decades. But what does it mean? And why is it relevant for automation software? Essentially, a language that is object-oriented provides three important programming mechanisms: encapsulation, inheritance, and polymorphism.³

Encapsulation is the ability of an object to maintain its own methods (or functions) and properties (or variables). For example, an “engine” object might contain methods for starting, stopping, or accelerating, along with properties for “RPM” and “Oil pressure”. Further, encapsulation allows an object to hide private data from any entity outside the object. The programmer can control access to the object’s data by marking methods or properties as public, protected, or private. This access control helps abstract away the inner workings of a class while making it obvious to a caller which methods and properties are intended to be used externally.

Inheritance allows one object to be a superset of another object. For example, one can create an object called `Automobile` that inherits from `Vehicle`. The `Automobile` object has access to all non-private methods and properties of `Vehicle` plus any additional methods or properties that makes it uniquely an automobile.

Polymorphism is an extremely powerful mechanism that allows various inherited objects to exhibit different behaviors when the same named method is invoked upon them. For example, let us say our `Vehicle` object contains a method called `CountWheels`. When we invoke this method on our `Automobile`, we learn that the `Automobile` has four wheels. However, when we call this method on an object called `Bus`, we find that the `Bus` has 10 wheels.

Together, encapsulation, inheritance, and polymorphism help promote code reuse, which is essential to meeting our requirement that the software package be flexible. A vendor can build up a comprehensive library of objects (a serial communications class, a state machine class, a device driver class, etc.) that can be reused across many different code modules. A typical control software vendor might have 100 device drivers. It would be a nightmare if for each of these drivers there were no building blocks for graphical user interface (GUI) or communications to build on. By building and maintaining a library of foundation objects, the vendor will save countless hours of programming and debugging time.

All three tenets of object-oriented programming are leveraged by the use of interfaces. An interface is essentially a specification that is used to facilitate communication

between software components, possibly written by different vendors. An interface says, “if your code follows this set of rules then my software component will be able to communicate with it.” In the next section we will see how interfaces make writing device drivers a much simpler task.

C++ and Device Drivers

In a flexible automation platform, one optimal use for interfaces is in device drivers. We would like our open-architecture software to provide a generic way for end users to write their own device drivers without having to divulge the secrets of our source code to them. To do this, we define a simplified C++ interface for a generic device, as shown here:

```
class IDevice
{
public:
    virtual string GetName() = 0; //Returns the name
                                //of the device
    virtual void Initialize() = 0; //Called to
                                //initialize the device
    virtual void Run() = 0; // Called to run the device
};
```

In the example above, a C++ class (or object) called `IDevice` has been defined. The prefix `I` in `IDevice` stands for “interface”. This class defines three public virtual methods: `GetName`, `Initialize`, and `Run`. The `virtual` keyword is what enables polymorphism, allowing the executing program to run the methods of the inheriting class. When a virtual method declaration is suffixed with `=0`, there is no base class implementation. Such a method is referred to as “pure virtual”. A class like `IDevice` that contains only pure virtual functions is known as an “abstract class”, or an “interface”. The `IDevice` definition, along with appropriate documentation, can be published to the user community, allowing developers to generate their own device drivers that implement the `IDevice` interface.

Suppose a thermal plate sealer manufacturer wants to write a driver that can be controlled by our software package. They would use inheritance to implement our `IDevice` interface and then override the methods to produce the desired behavior:

```
class CSealer : public IDevice
{
public:
    virtual string GetName() {return ‘‘Sealer’’;}
    virtual void Initialize() {InitializeSealer();}
    virtual void Run() {RunSealCycle();}
private:
    void InitializeSealer();
    void RunSealCycle();
};
```

Here the user has created a new class called `CSealer` that inherits from the `IDevice` interface. The public methods, those that are accessible from outside of the class, are the interface methods defined in `IDevice`. One, `GetName`, simply returns the name of the device type that this driver controls. The other methods, `Initialize()` and `Run()`, call private methods that actually perform the work. Notice how the `private` keyword is used to prevent external objects from calling `InitializeSealer()` and `RunSealCycle()` directly. When the controlling software executes, polymorphism will be used at runtime to call the `GetName`, `Initialize`, and `Run` methods in the `CSealer` object, allowing the device defined therein to be controlled.

```
DoSomeWork()
{
    //Get a reference to the device driver we want to use
    IDevice &device = GetDeviceDriver();

    //Tell the world what we're about to do...
    cout << "Initializing " << device.GetName();

    //Initialize the device
    device.Initialize();

    //Tell the world what we're about to do...
    cout << "Running a cycle on " <<
    device.GetName();

    //Away we go!
    device.Run();
}
```

The code snippet above shows how the `IDevice` interface can be used to generically control a device. If `GetDeviceDriver` returns a reference to a `CSealer` object, then `DoSomeWork` will control sealers. If `GetDeviceDriver` returns a reference to a pipettor, then `DoSomeWork` will control pipettors. Although this is a simplified example, it is straightforward to imagine how the use of interfaces and polymorphism can lead to great economies of scale in controller software development.

Additional interfaces can be generated along the same lines as `IDevice`. For example, an interface perhaps called `ILMS` could be used to facilitate communication to and from a `LIMS`.

The astute reader will notice that the claim that any third party can develop drivers simply by implementing the `IDevice` interface is slightly flawed. The problem is that any driver that the user writes, like `CSealer`, would have to be linked directly to the controlling software's executable to be used. This problem is solved by a number of existing technologies, including Microsoft's `COM`⁴ or `.NET`⁵, or by `CORBA`⁶. All of these technologies allow end users to implement abstract interfaces in standalone components that can be linked at runtime rather than at design time. The details are beyond the scope of this article.

Event-Driven Scheduling

Before discussing scheduling, we must first discuss the concept of threading. A thread is one part of a program that executes on a microprocessor.⁷ When a microprocessor executes a single-threaded program, it runs only one part of that program at any given time. However, in a multithreaded program, the microprocessor is able to execute several parts of the program simultaneously, potentially providing great performance and scalability advantages. For example, a multithreaded scheduler can have different threads responsible for communicating with each device on the system and another thread responsible for reading the mouse and keyboard. If a device stops communicating, only the one thread responsible for that device has to facilitate some sort of recovery. In contrast, the one execution thread of single-threaded program has to do all of the scheduling, device communication, and user interaction. In this case, careful programming is required to prevent the entire application from being adversely affected if difficulties are encountered communicating with a single device. Multithreaded programs also have the advantage that different threads can execute concurrently on different microprocessors in a multiprocessor system. A program with two threads will therefore execute faster on a dual-processor system than on a single-processor system. A single-threaded program, however, will not experience much, if any, speed benefit from a computer that has multiple microprocessors.

There are many types of schedulers in existence that manage large numbers of devices in integrated systems.⁸ Most automation software uses something called a "static scheduler". With most static scheduling programs, the user manipulates the software's GUI to define the steps of an assay to run and then presses the "start" button. This causes the static scheduling algorithm to translate the user's method into a linear series of steps that will be executed at preset times. When running, the automation software requires only a single thread of execution to issue commands to the devices in linear fashion. While this may add predictability to a run, throughput and system robustness are sacrificed in most cases.

Event-driven controllers use multithreading to maximize throughput and robustness. These controllers can use multiple threads of execution to monitor the activities of each device in the system. As soon as a device has finished operating on its sample, the controller can immediately command a robot to move the sample on to the next device in its process. If one device goes offline because of an error, the others can continue processing if the user so desires. If there are two identical readers in the system and one goes offline, the controller can continue processing all samples on the first reader instead. This robust error behavior is extremely difficult to reproduce on static schedulers.

A deadlock condition occurs when a scheduler gets into a predicament that it cannot solve without some kind of user intervention. For example, a sample at station A needs to

move to station B, but the sample at station B needs to go to station A. If there is no available intermediate location to move either sample, then a deadlock occurs. Static schedulers never suffer from deadlocks because deadlocking schedules can be omitted from consideration by the scheduling algorithm. However, event-driven controllers need to be “taught” how to avoid deadlocks, putting an additional burden on the programmer or end user.

	Static scheduler	Event-driven controller
When an error occurs...	All processing halts	Processing can continue or halt as desired
Device pooling	Can improve throughput but does not provide redundancy	Can improve throughput and provide redundancy
Throughput	Cannot be maximized unless the exact timings of each operation are previously known. The scheduler doesn't know when the operations actually complete, so samples typically cannot be moved right away	Can be maximized because samples can be picked up as soon as an operation is complete
Startup time	Slower. Algorithm must calculate static schedule before run can begin	Very fast. No startup calculations required
Time constraints	Can be exact	Not exact, but can be maintained to within a few seconds
Deadlock conditions	Impossible	Possible

It is much easier to implement an event-driven controller in a multithreaded language such as C++. Languages such as Visual BASIC 6.0 do not provide the ability to multithread and therefore are not ideal candidates for this architecture.

C++, Templates, and Code Optimization

One of the more advanced features of C++ is templates. Templates allow the same code to be reused for object of different types.

```
SortSomeLists
{
    //Create a list of integers
    list<int> myints;
```

```
//Push 3 integers into our list
myints.push_back(1);
myints.push_back(10);
myints.push_back(5);

//Sort them
sort(myints.begin(), myints.end());

//Now create a list of floating-point numbers
list<float> myfloats;

//Push in some values
myfloats.push_back(12.3);
myfloats.push_back(10.9);
myfloats.push_back(50.4);

//...and sort. Notice we used the same sort function
to sort the integers and the floats
sort(myfloats.begin(), myfloats.end());
}
```

The existence of templates in the C++ language has led to the development of many reusable, optimized algorithms. Several of these are available with the Standard Template Library, a standard component of C++. In addition to implementing containers (such as map, linked list, and vector), this collection of classes and functions provides extremely optimized versions of sort, difference, union, find, and many others. Because they are template-based, the same algorithms can be used to sort strings, integers, floats, or even objects of user-defined type. For example, there is no need to implement a different sorting algorithm for each data type. Templates, therefore, are another tool in the C++ programmer's toolbox that enables the maximization of code reuse. As a result, development time is reduced while bugs are minimized.

Some newer programming languages, such as JAVA⁹ and C#, are object-oriented like C++, but add additional features such as garbage collection and cross-platform portability. Garbage collection is a tool that frees the programmer from having explicitly manage memory. The execution environment runs a thread that is responsible for determining when allocated memory is no longer needed. When this garbage collector finds memory that is safe to delete, it asynchronously frees it up, allowing the memory to be reused for other purposes. The problem is that if the garbage collector chooses to free up a large chunk of memory at an inopportune time, unpredictable performance issues may arise with the application. Because C++ has no automatic garbage collector, programmers must manage their own memory allocation and deallocation. Therefore, memory can always be freed when it is safe and convenient to do so.

Cross-platform portability is the ability of the same program to execute on multiple operating systems. Unlike C++, which gets compiled directly into machine-executable instructions, languages like JAVA instead get compiled into an intermediate form called byte-codes. These byte-codes are translated at runtime into machine-executable codes by

a program called a virtual machine. Each different operating system can have its own JAVA virtual machine. Therefore, code written on one platform can execute on any other platform provided a virtual machine is available for that operating system. This is an obvious advantage for a vendor because it means software written in JAVA can execute on a number of different operating systems without having to do any porting. One rather large disadvantage, however, is that JAVA code typically executes slower than equivalent C++ code because the byte-codes have to be translated into machine code at runtime. Because performance is a primary concern for an event-driven scheduler, this makes languages such as JAVA and C# less desirable.

CONCLUSIONS AND SUMMARY

C++ is an object-oriented language that provides several advantages in developing automation control software. Encapsulation, inheritance, and polymorphism provide programmers the ability to develop and implement interfaces. As a result, software can be made much more scalable and flexible than it otherwise could. C++ allows for multithreading, which provides the possibility of implementing an event-driven controller. This architecture provides far higher throughput and robustness than typical static schedulers. And C++'s templates provide a mechanism to reuse

algorithms across different data types. There are countless programming languages out there, many of which would be good candidates for writing automation control software. C++, however, being an object-oriented language that provides for multithreading and template-based coding, is an ideal choice.

REFERENCES

1. Paszko, Christine *Laboratory Information Management Systems*. 2nd edition. New York, NY: Marcel Dekker, Inc.; 2001.
2. Stroustrup, Bjarne *The C++ Programming Language*. 3rd edition. Reading, MA: Addison Wesley Professional; 1997.
3. Coad, Peter; Nicola, Jill *Object-Oriented Programming*. 1st edition. Upper Saddle River, NJ: Prentice Hall; 1993.
4. Rogerson, Dale *Inside COM Microsoft Programming Series*, Microsoft Press: Redmond, WA; 1997.
5. Platt, David S. *Introducing Microsoft.NET*. 3rd edition. Redmond, WA: Microsoft Press; 2003.
6. Bolton, Finton *Pure CORBA*. 1st edition. Indianapolis, IN: Sam's; 2001.
7. Beveridge, Jim; Wiener, Robert *Multithreading Applications in Win32: The Complete Guide to Threads*. Reading, MA: Addison Wesley Professional; 1996.
8. Pinedo, Michael *Scheduling Theory, Algorithms, and Systems*. 2nd edition. Upper Saddle River, NJ: Prentice Hall; 2002.
9. Flanagan, David *JAVA in a Nutshell* 5th edition O'Reilly Media: Sebastopol, CA; 2005.