An Interface Between Object-Oriented Systems

Lawrence A. Crowl

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 211

April 1987

Abstract

The description 'object-oriented' may apply to both programming languages and operating systems. However, creating an interface between an object-oriented programming language and an object-oriented operating system is not necessarily a straightforward task. Chrysalis++ is a C++ interface to the Chrysalis operating system for the BBN Butterfly Parallel Processor. The development of Chrysalis++ highlights strengths and weaknesses of C++ and the problems of adapting a language based on a conventional memory model to a shared memory parallel processor.

This work was supported by United States Army Engineering Topographic Laboratories research contract number DACA76-85-C-0001 and National Science Foundation Coordinated Experimental Research grant number DCR-8320136. We thank the Xerox Corporation University Grants Program for providing equipment used in the preparation of this paper.

1 Introduction

The Chrysalis operating system for the Butterfly Parallel Processor presents an object-oriented programming environment based on shared memory [BBN, 1985a; BBN, 1985b]. However, because of Chrysalis's low level orientation and its use of type-unsafe features of the C programming language [Kernighan and Ritchie, 1978], programs using the environment are difficult to program and highly error-prone.

Using C as the primary programming language for the Butterfly does not fully realize the benefit of Chrysalis's object orientation. An object-oriented programming language is a natural candidate for improving the Chrysalis environment. The C++ programming language [Stroustrup, 1986] provides a number of advantages in developing such an interface. First, C++ uses C as its object language. Since C is available for the Butterfly, less effort is necessary to implement C++ on the Butterfly than to implement another language from scratch. Second, C++ is a static language. This allows C++ programs to be comparable in run-time efficiency with more conventional programming languages. Run-time efficiency is important, since the reason for using a parallel processor is speed, not convenience. Third, compatibility with C allows programmers to work around any potential inadequacies in C++, at least relative to what programmers expect.

This paper reports the successes and problems encountered in the development of Chrysalis++ [Crowl, 1986], a C++ interface to Chrysalis. The development of Chrysalis++ uncovered many strengths and weaknesses in C++. Some apply to C++ in general, others apply only to its adaptation to a parallel programming environment. It is important to note that C++ is a sequential language; its use in a parallel programming environment is therefore outside the bounds of its design.

1.1 C++ Language Overview

The C++ programming language is a (mostly compatible) superset of the C programming language. It is statically typed and object-oriented. This subsection is an overview of the most important features, that exist in C++, but not in the C language. Refer to [Stroustrup, 1986] for a more complete description.

C++ encapsulates data within objects. Each object is an instance of a class. The class statically defines the component variables and types of each instance. It also defines the member functions and operations that may be performed on instances of the class. The expression `o.f( a )` invokes the member function `f` of the object `o` with the argument `a`.

C++ provides single inheritance. The programmer may derive new classes from existing base classes. The derived class inherits the variables and operations of the base class. The programmer defines which variables and operations of the class may be used outside of the definition of the class.

Creation of instances of a class automatically invokes a member function called a *constructor*, and deletion automatically invokes a member function called a *destructor*. Since creation and deletion are integral parts of declaring a variable, these functions allow explicit programmer control over all phases of an object's lifetime. For example, in defining a class `T`, the member function definition

```
T( int i ) { /* ... */ }
```

defines a constructor for class `T` taking a single integer argument. The definition

```
T x( i );
```

declares a variable `x` containing[1] instances of class `T` and calls the constructor for the class passing the integer argument `i`.

C++ provides a mechanism for generic classes based on C preprocessor macros. The generic class is defined as a macro that takes the parameter class as an argument. The instantiation of a generic class results in a specific class declaration. Programmers then use objects defined with this specific class declaration. Basing the generic class mechanism on the C preprocessor causes several problems,[2] which leads to the conclusion that the C++ compiler should implement generic classes directly.

The `new` and `delete` operators create and destroy objects in "free store" without explicit object size calculations, replacing calls to `malloc()` and `free()`. These operators automatically call the constructors and destructors for the appropriate type. For example, the following declares a pointer to an instance of class `T` and initializes it to point to a new instance of `T` in free store.

```
T *p = new T( i );
```

C++ provides inline function expansion. This capability allows one to write highly modular code without sacrificing performance using called functions or semantics using preprocessor macros.[3]

C++ provides for default parameters. For instance, the function definition

```
int func( int a, int b = 3 ) { /* ... */ }
```

indicates a default second parameter and interprets

`func( 8 );` as `func( 8, 3 );`

The standard C operators (`+`, `-`, `*=`, etc.) and programmer-defined function names may be overloaded. Overloaded operators and names are distinguished by argument type. This allows more concise, notationally convenient program code.

The type casting facility may also be overloaded. This allows the programmer to introduce procedures to explicitly reform data, as opposed to just re-interpreting the representations. For instance, the programmer may define a procedure that accepts an instance

[1] In contrast to Smalltalk and CLU, C++ variables are not references to objects, but contain objects as values.

[2] The generic macros have greater restrictions on spacing in the source code because they construct identifiers. In addition, one cannot use the // comment form in multiline generics because the preprocessor concatenates lines into a single line before the C++ compiler sees them. Therefore, everything after the first // in the macro is commented out.

[3] Functions pass parameters by value, macros pass parameters by name.

of a class and returns an integer and have the procedure executed when the cast `(int)` is applied to an instance of the class.

In addition to standard C pointers, C++ has *references*. References are defined by the prefix use of the `&` operator. The programmer does not indicate explicit dereferencing, the compiler dereferences as appropriate. For example,

```
void func( int &a ) { a += 2; }      with      { int c; func( c ); }
```

passes `c` "by reference".

1.2 The Butterfly and Chrysalis

The Butterfly contains up to 256 processor nodes interconnected by an $O(n \log n)$ switching network. Each node contains a Motorola 68000 microprocessor, up to 4 megabytes of memory, and a microcoded processor node controller (PNC). The PNC provides address translation, transparent access to remote memory using the switching network, and microcode support for process scheduling, block memory copy and atomic operations.

The Chrysalis operating system is very low level. It has no file system, but does provide access to a host system to load programs and read and write sequential files.

Chrysalis provides support for heavyweight processes compiled as separate programs. Each process exists on exactly one processor for the lifetime of the process. By convention, the code and activation stack reside on the same node as the process. The data often resides on a different processor. Process startup is expensive, but the process context switch time is small.

Chrysalis refers to every system object, including processes, with a unique 32-bit object identifier (`OID`). An object may own other objects. By default, the process object that creates an object owns that object. When an object is deleted (or a process terminates), any objects it owns are also deleted.

Chrysalis allocates each processor node's local memory into a number of memory objects. Operations exist to create, delete, map and unmap a memory object. The map operation places any memory object on the system into the 24-bit virtual address space of the process performing the map. Delete operations on memory objects are delayed until no process has the memory mapped in.

Chrysalis provides microcode support for several atomic operations. These include atomic update operations on 16-bit integers and atomic queueing operations for 32-bit values. The atomic integer operations take up to six times as long as equivalent non-atomic operations.

1.3 The Problem

The Chrysalis design allows explicit programmer control over system operations. The interface provided for Chrysalis uses the C programming language. This environment requires the user to write a substantial amount of code that relies heavily on the explicit use of pointers, size calculations and type casting, which in turn severely reduces (the already minimal) compiler type checking. As a result, the programming process is error-prone. Since most programming mistakes on the Butterfly manifest themselves as "bus errors", tracking down these mistakes can be extremely time consuming.

Programming under Chrysalis is also cumbersome. For instance, Chrysalis processes require as much as a hundred lines of user code to start. Process startup code involves building two records (most of the fields are given default values), and calling up to five procedures. The atomic queue mechanism requires the programmer to allocate the queue buffer space before creating the queue itself. Each queue operation takes two parameters which are almost always given default values. However, the programmers must remember to provide them.

Because of the difficulty in writing C programs that use Chrysalis, programmers tend to avoid it. Efforts to improve the Butterfly's programming environment include the programming languages Lisp [Steinberg *et al.*, 1986], Lynx [Scott, 1987; Scott, 1986] and Modula-2 [Olson, 1986], and the library packages NET [Hinkelman, 1986], SMP [LeBlanc *et al.*, 1986], and the Uniform System [BBN, 1985c]. With the exception of Modula-2, these efforts all provide a model of computation different from the Chrysalis operating system. Chrysalis++ represents another effort to hide the complexity of the Chrysalis environment, this time using object-oriented programming in C++. In contrast to the earlier efforts, Chrysalis++ provides the same model of computation and mechanisms as Chrysalis. In addition, Chrysalis++ is nearly as efficient as the bare Chrysalis and C mechanism.

1.4 The Solution

The goal of Chrysalis++ is to provide a much simpler and safer interface to Chrysalis than that provided by C. It achieves this goal by using the object-oriented features of C++, drastically reducing the coding interface between the user and the system. Chrysalis++ is not an attempt to build a "new" system and implement it on Chrysalis.

The constraints on the development of Chrysalis++ were to avoid modifying the C++ compiler (and hence the language) and to avoid modifying the Chrysalis operating system. The C++ runtime support was modified in the process of porting it from Unix to Chrysalis.[4]

Chrysalis++ is not a full replacement for the standard Chrysalis and C environment. It provides only those features that are commonly needed at the user level. If the Chrysalis++ mechanisms prove inadequate for certain high-performance applications, programmers may use the basic Chrysalis operations directly. More importantly, programmers may define new mechanisms to provide the needed function. This approach maintains the benefits and improves the functionality of Chrysalis++.

The automatic calling of class constructors and destructors in C++ provides a powerful mechanism for insuring proper creation and destruction of data objects. Since the principal difficulty in the Chrysalis environment is creation, access, and destruction of Chrysalis objects, defining C++ classes to control access to Chrysalis objects creates a better programming environment. It greatly reduces the amount of coding required by the user and

[4] This was an interesting experience in its own right, primarily because Chrysalis functions are not identical to their Unix counterparts.

simultaneously increases the likelihood of correct code at run time because programmers cannot improperly create or access Chrysalis objects. In addition, using C++ classes to control access to Chrysalis objects binds access to an object to the lifetime of the variable used to represent it. This binding represents a conceptual cleanliness because users do not have to keep track of whether or not a variable references a valid object: it always refers to a valid object.

Casting explicit Chrysalis object management into into implicit C++ object management provides the Chrysalis++ programming environment with a single object management strategy. This strategy is successful in reducing user code to manage various types of Chrysalis objects from between six and a hundred lines of largely type-unsafe C code to a single line of type-safe C++ code.

2 Typed References to System Objects

Chrysalis provides no direct process-to-process communication facility. Running processes communicate by passing each other object references (`OID`s). Each process then interacts with the object referenced. There are two problems with `OID`s. First, they are identical to integers under the C system of structural type equivalence. Therefore, programmers may only maintain `OID` identity by convention. Second, `OID`s carry no type information. The substitution of an `OID` referencing an object of one type for an `OID` referencing an object of another type is a difficult error to track down.

Chrysalis++ does not use `OID`s directly because C++ also considers `OID`s and integers to be equivalent types.[5] As a result, any classes that create or access Chrysalis objects cannot overload functions based on the distinction between an `OID` and an integer. Since this capacity is desirable, a new class, called `handle`, serves as the means of referencing Chrysalis objects. This class is distinguishable from integers.[6]

Chrysalis processes are compiled separately, so there is no opportunity to ensure consistent use of data types between processes with the C++ typing mechanism. Instead, Chrysalis++ provides a run time solution. Since `OID`s and handles are too small to contain type information, there is the potential for a handle for a class of one type to be interpreted as a handle for a class of another type in a different process. Chrysalis++ solves this problem by hashing the type name of an object into a byte code and then tagging the code with the object when it is created. A constructor setting up access to such an object will check the expected type code of the object against the actual type code stored with the object.

There are some drawbacks to this mechanism. First, the hashing mechanism may map two different type names to the same code, although the probability is low. Second, type hashing is not fully compatible with the C++ type scheme because it is based on the type name. For instance, a typedef from one type to another will generate different codes even though they are the same type under C++. Third, spacing problems occur with the type

[5] This is based on the notion of a 'typedef' as an alias for a previous type. Its presence in C++ is necessary for compatibility with C.

[6] A type cast operation between handles and `OID`s allows Chrysalis++ code to use the Chrysalis functions that take `OID`s.

hashing mechanism because the C preprocessor is used to convert the type name into a C++ manipulable string. Ideally the program needs access to the compiler's notion of type identity in order to be fully compatible with the C++ type scheme and avoid the preprocessor. When fully integrated into the compiler, the type hashing scheme can be effective [Scott and Finkel, 1984].

3 Processes

Butterfly programs consist of many processes. The program development environment considers each type of process to be a 'program' in the Unix or C sense. Programs start with the execution of a single 'parent' process as defined in a load image file. This process then creates the remaining 'child' processes.

Most operating systems provide a mechanism for killing an errant program by killing the process associated with the program. On the Butterfly, there are potentially hundreds of processes in any given program, so killing each process manually is extremely tedious. Chrysalis solves this problem by allowing the parent process to retain ownership over child processes. When the user kills the parent process, the system will delete (kill) all the child processes and memory objects the parent owns. Termination of a process is the same as deletion, so programmers must explicitly wait on the completion of child processes in order to prevent the parent's termination from prematurely killing its children.

3.1 Single Processes

Chrysalis++ represents all system objects with C++ class variables. In the case of processes, these are represented by 'process variables'. In keeping with the policy of linking object and variable lifetimes, a program should not leave the scope of a process variable until that process finishes. So, each process declared as a static or local variable is an owned, or child, process. The creator process will automatically wait on completion of the child process before exiting the scope enclosing the process variable. However, process variables allocated out of free store (i.e. with the `new` operator) have no scope associated with them. As a consequence, process variables created in free store represent disowned, or independent, processes which will continue to execute after the creating process completes. The ability of a C++ class to determine its allocation makes this possible. The user may explicitly wait for completion of a process in free store by applying the `delete` operator to it.[7]

Processes under Chrysalis++ require five parameters for creation: the file name of the program to execute as a process, the argc/argv parameters to the process, the processor upon which to execute the process, and the number of address mapping registers allocated to the process.[8] The last two parameters have default values which C++ will provide if the programmer does not. This reduces the burden on programmers with simple needs.

[7] Other mechanisms exist to abort a process prematurely.

[8] The address mapping registers are a critical resource on the Butterfly, and generally must be programmer specified.

Processes may obtain arguments through either an argc/argv mechanism (similar to the Unix/C argc/argv mechanism) or through the system's global name server. The initial implementation of process creation in Chrysalis++ required passing explicit argc/argv values. It soon became clear that a much simpler argument mechanism would satisfy the majority of processes. The final implementation also allows a single argument string that the process constructor parses into the appropriate argc/argv values much as the command shell parses program arguments. The forms are distinguished using the overloading features of C++.

Since programs pass many object handles through the argc/argv mechanism, Chrysalis++ provides type casts between handles and character strings to aid in convenient passing of handles through this mechanism. This facility is critically dependent on the C++ ability to define programmer implemented type casts.

3.2 Multiple Processes

Since programs usually start processes in groups of the same kind, a mechanism for creating multiple processes aids the programmer considerably. Programmers generally allocate processes within a group to different processors to balance the computational load. This subsection discusses three approaches to implementing multiple processes: process arrays, process pointer arrays, and a multiple process class. The first two approaches do not work; Chrysalis++ adopts the third approach.

Process Arrays

The first approach to multiple processes is to allocate an array of processes. This is what naturally comes to mind when considering a set of processes of the same type. Unfortunately, this approach has two severe problems.

The first problem is that, using C++, one can allocate an array of class instances only when the class has a constructor taking no arguments. However, constructors for processes require at least the name of the program to execute. Certainly, one should be able to specify the parameters for constructors when allocating an array of class instances. Even constant parameters would be better than none at all. A more useful mechanism would allow passing the index value to the constructor for each element within the array.

The second problem is that the number of elements in local arrays is determined at compile time. This would restrict process arrays to a fixed number of processes. This is unacceptable for many programs where the number of processes is matched to the number of processors available at run time. C++ should support determining the number of elements in a local (i.e. automatic) array upon allocation. This would require the compiler to implement a level of indirection transparent to the user.[9]

[9] The preferable place to allocate such arrays is on the stack. However, using C as a target language will not permit this, so the compiler must generate code to explicitly allocate and free the space for the array using `malloc()` and `free()`.

Process Pointer Arrays

The second approach to managing multiple processes is to have the programmer construct a process group using an array of pointers to processes allocated out of free store. In the case of a single process, Chrysalis++ defines process variables allocated out of free store to be disowned (i.e. independent) processes. If the programmer maintains pointers to processes, the parent process would exit independently of the completion of the child processes unless the user specifically waited for the completion of each of the child processes. Also, if the user kills the parent process while some of these disowned processes are running, the system will not automatically delete them. The user would have to kill each process manually. In addition, the amount of user code required for this solution is unacceptable.

3.3 Multiple Process Class

The third approach has a class that represents multiple child processes with a single class instance. Chrysalis++ adopts this approach. The class uses the same free store distinction in determining child process ownership as used in the single process class. The constructor the class automatically allocates the requested number of the child processes. The destructor waits for the completion of each child process. This approach has a side benefit of increasing the efficiency of creating multiple processes because the implementation does some operations only once instead of once for each process.

4 Shared Objects

Processes create and share memory objects. The creation of memory objects in Chrysalis is a four step process consisting of determining object size in bytes, creating the memory object (which takes the size and returns the `OID` of the object), mapping the object into the process's address space (which takes the `OID` and returns a pointer to the memory), and initializing the memory object. The size computation and initialization often take a user parameter to size the object dynamically. Processes access existing memory objects by obtaining the `OID`s and mapping the associated objects into the their address space.

Programs generally cannot have all memory objects they reference mapped in at all times because the hardware address mapping registers are a scarce resource. Since the mapping operation is too expensive to perform for each use of the memory object, automatic management of the mapping registers is a non-trivial task. Programmers must explicitly manage the duration over which the object is mapped in.

The treatment of memory objects shared among many processes is the heart of Chrysalis++. These memory objects are most naturally represented as the sharing of instances of user-defined classes. The development of shared objects defined the critical concepts in Chrysalis++.

However, to use the class mechanism, the program must distinguish between creation of a new object and access to an existing object. Chrysalis++ achieves the distinction between these two operations by overloading the constructor operation. When the first parameter to a variable constructor is an object handle, the variable represents access to the existing object referenced by the handle. Otherwise, the constructor represents creation of a new object.

4.1 Controlling Initialization

The main problem with shared objects lies not in distinguishing between the two constructors, but in controlling invocation of the user's object initialization code and in integrating system-supplied addresses into the object mechanism. This problem arises because C++ has no notion of an object entering (or leaving) an address space. All C++ objects live and die within the program's address space.

Three failed approaches to controlling initialization were a new object allocator, a base class for shared object, and deriving shared objects from user classes. The approach adopted is a class that acts as a pointer to shared objects.

Another Allocator

The most obvious approach to shared objects is to have a variant of the `new` operator that allocates a memory object instead of going to `malloc()`. This approach has three problems.

The first problem is that one must redefine the `new` operator for all uses or none. There is no mechanism for indicating which `new` operator to use. Since Chrysalis++ programs must still use the normal free store allocation, Chrysalis++ could not redefine the allocation to work for shared objects. The C++ language should provide a mechanism for multiple storage allocators.

The second problem with this approach is that memory objects are usually dynamically sized to meet specific program needs. However, C++ classes are statically sized. Any solution forcing static sizing is unacceptable. This problem can be solved with the concept of an open class. An open class is one in which the last component is an array whose number of elements is not bound until allocation. Open structures are simulated in C by defining a structure with a one element array as its last component and then requesting the space appropriate to the number of elements actually needed from `malloc()`.

Computing the number of elements needed in the tail array of an open class is best left to the class itself. This in turn requires the ability to do some computation in the constructor before the actual allocation of space. Of course, one cannot use variables of the class instance in this computation. In conjunction with the multiple allocator assignment, size precomputation in the constructor implies passing the allocator as a parameter.

The third problem is that shared objects have no scope associated with them since they use the free store mechanism. This means the distinction between owned and disowned objects cannot be made on this basis.

Shared Base Class

The second approach derives the user's class from a system provided base class that performs the Chrysalis memory object operations. Unfortunately, the storage space for the instance is allocated before the base class constructor has a chance to indicate where it should be allocated. The derived class can indicate where the instance is to be placed, but not the base class.

An additional problem is that the language always calls the user's constructor for the derived class to initialize the object instance. The user's constructor must make the distinction between creation and access to an existing object. This is not something the user should have to code.

Shared Derived Class

Since the major problem with the shared base class approach was that the derived class is called first, the third approach derives the shared class from the user's class and intercepts the allocation mechanism. Unfortunately, this only allows statically sized instances because the implementation places derived class data storage immediately above base class data storage. The C programming trick of allocating more space than a structure calls for and just using the space beyond the declared bound of the structure will not work. Since many of the objects shared under Chrysalis are dynamically sized, this was unacceptable.

This method forces all objects to be allocated with the `new` operator because static and local (automatic) variables cannot sensibly modify their addresses. This in turn implies that Chrysalis++ cannot use the free store distinction to control ownership of the object.

Again, the user's constructor will be called at all times and it must make the distinction between creation and access to an existing object.

Shared Class as a Pointer

The problem with the previous two approaches was the automatic calling of the constructor. So, the fourth and final approach uses a class containing a pointer to the shared object. Since the constructor for the referent of a pointer is not automatically called, the sharing mechanism can call the constructor explicitly when desired.

This allows calling a user defined function to determine the size of the object, and then request the appropriate space from Chrysalis. Unfortunately, there is no mechanism for indicating at what address the instance is to be initialized.[10] The ability to specify where an object is to be created would also be useful for device drivers that have input and output registers at specific addresses. The solution to these problems is to require the user's class to provide a `size()` member function to compute the amount of space needed and a `init()` member function to initialize the instance. The constructor is not called.

The shared class as pointer approach allows the use of the free store distinction in determining object ownership.

Chrysalis++ overloads the unary prefix * operator to allow the shared class to look like a pointer. Dereferencing the shared class is signaled with a prefix * just as it is in C. In standard C, the expression `*(a).b` is equivalent to `a->b`. However, in C++ the definition of

[10] One can pass a pointer to the constructor and have the object initialize itself at that location. This method requires the cooperation of the class in handling explicit pointer operations.

an overloaded unary prefix `*` operator does not also allow the overloaded use of the binary infix `->` operator. Thus, only the former notation is allowed.

4.2 Generic Shared Class

Each shared user class requires a different 'pointer class'. Since the code to implement this class is precisely the code that is difficult to get right, programmers do not want to code it for each shared class. This problem is solved by taking advantage of the generic class mechanism described in [Stroustrup, 1986, pages 209–210].

The drawback is that the definition of generic classes generally implies that the programmer knows the parameters of the base class functions. For instance, to allow dynamic sizing of shared objects, Chrysalis++ must have a parameter to the `size()` and `init()` member functions. However, not all shared objects will actually use the parameter. A more general approach would allow generic functions with an unspecified number of arguments that will in turn be passed to the appropriate parameter class functions. The programmer need not assume, for instance, that the parameter class constructor takes a single integer argument. This is the assumption made in the definition of the shared generic class.

4.3 Delayed Delete

Deletion of Chrysalis memory objects is delayed until no process has the object mapped into its address space. The system cannot invoke the user's destructor for a shared object because another process may have the object mapped into its address space. The other process will continue to use the memory with its now invalid data structure. Because of these delayed delete semantics, the user's shared class destructor will never be called. This is an instance of a non-object-oriented result arising from creating an interface between an object-oriented programming language and an object-oriented operating system. This highlights a fundamental question concerning the meaning of shared objects in a parallel programming environment. When does a delete take effect?

5 Atomic Integers

Chrysalis provides one atomic integer operation, called *clear-Then-add* or *cTa*, which takes three parameters, a pointer to a 16-bit word, a 16-bit mask and a 16-bit value. The operation clears the bits in the word corresponding to the bit set in the mask, then adds the value into the word. This operation has degenerate forms for the operations *and*, *inclusive or*, and *add*.

Under Chrysalis, the atomic operation is a special function that users may invoke on any memory word, there is no notion of an 'atomic' variable. Non-atomic operators may be applied directly to variables intended to be used in an atomic fashion.

Chrysalis++ uses the C++ class definition and operator overloading facilities to force all operations on variables intended to be atomic to be atomic. All non-assignment operators are atomic in the sense that they atomically read the value. Assignment operators that are atomic are `=`, `+=`, `&=` and `|=`. The standard C++ assignment operators that cannot be atomic are `*=`, `/=`, `%=`, `^=`, `<<=` and `>>=`. These operations are not defined for atomic integers and will generate a compile-time error if used. Note that while individual operations are atomic, combinations of these operations are not atomic. Atomic operations also work only on short integers.

The Chrysalis functions return the old value of the word. This capability is vital in implementing test and set, array index allocation, and other atomic operations. Since the C assignment operators return the new value, the equivalent atomic assignment operators could not return the old value. Otherwise, the semantics of the atomic assignment operations would be counter-intuitive. The result is a set of member functions on atomic variables (`add()`, `sub()`, `and()`, `ior()` and `cTa()`) that return the old values.

The original plan was to derive the atomic short integer class from the short integers. However, since primitive C++ types, such as short integer, are not classes, they cannot serve as the base for derived classes. This is unfortunate, since many of the operations are the same. However, since the semantics are simple a class definition from scratch is reasonably short. Atomic integer class definition uses inline functions and is as efficient as the Chrysalis mechanism.

C++ does not distinguish between prefix and postfix use of an overloaded `++` or `--` operator. This means that use of these operators would be restricted to one action or the other, but not both. Rather than provide a definition where one use provides unexpected results, both uses are illegal in Chrysalis++. This is not a serious problem because there are simple equivalent syntactic forms.

6 Atomic Queues

Chrysalis supports atomic queue operations on 32-bit values, usually integers or object handles. These queues buffer values enqueued or handles of processes waiting to dequeue a value. Chrysalis supports two forms of enqueue and dequeue. The waiting form causes the process to wait for successful completion; the polling form attempts the operation and returns an indication of success.

Under Chrysalis, the programmer must explicitly allocate the queue buffer memory before creating the queue itself. The Chrysalis++ form of the atomic queues wraps the queue buffer allocation into the creation of the queue itself.

The initial queue approach provided a separate queue type for each of the primary types of enqueued values: integers and object handles. This forces users to abandon type checking with explicit casting when enqueuing objects of different types. The solution implements queues as a generic class taking the type of the element to be enqueued as an argument. The Chrysalis++ generic queue mechanism will accept no class larger than 32 bits because of the Chrysalis limit of 32 bits on queue values. Because of the internal implementation, overloading the address of (`&`) or assignment (`=`) operators of the parameter class will produce unpredictable results.

A queue is potentially shared by many processes, so it should have the same delete semantics as shared objects. That is, delete the queue when the last user relinquishes access as opposed to when the original creator relinquishes access. Unfortunately, the run-time cost to implement this under Chrysalis is substantial, so it was not done. Memory objects and queues have different delete semantics. Object-oriented systems should carefully consider object management semantics and apply them uniformly.

7 Conclusions

Overall, the use of C++ to create an interface to Chrysalis is a success. However, this success was not as complete as hoped, primarily because C++ has no notion of an object entering an address space intact. It will need such a notion to be fully integrated into any system that shares memory but does not share a uniform virtual address space.

Several features of C++ helped in the development of Chrysalis++.

- Classes provided a clear organizational tool.
- The automatic invocation of constructors and destructors was the one feature that contributed most to user code reduction. The constructors handle the system object creation mechanism and the destructors automatically clean up.
- The ability to distinguish free store allocation of objects from other uses enabled the owned/disowned object distinction to be expressed concisely. The lifetime of system objects is equivalent to the lifetime of the variables used to represent them.
- Overloaded operations and function names kept the linguistic cost of manipulating objects low. Programmers do not have to remember a different set of operators for atomic integers as opposed to normal integers.
- User-defined cast operations allowed convenient value conversions. A re-interpretation of the representation would not have been appropriate in many cases. For instance, casting from an object handle to a character string requires the allocation and initialization of a string. This is something that would not have been possible without user-defined casts.
- The C++ default parameter mechanism allows the programmer to ignore those parameters for which the default is acceptable. Chrysalis uses default values a great deal. This is especially helpful for those features that must be specified on sufficient occasions to warrant inclusion in the interface, but are not necessary for simpler cases.
- Inline functions permit abstract operations to be implemented efficiently. Some operations would have been too expensive to abstract if a procedure call was necessary. For example, the atomic integer and queue operations would have been far too expensive if encased in a called procedure.
- Generic classes reduced programmer coding of direct, and error-prone, interactions with Chrysalis. This allows the system to provide the specific mechanisms for a given class automatically.

The C++ inheritance mechanism was of helped little in the development of Chrysalis++. Inheritance from primitive types would have increased its utility, but not enough to have a major influence. However, the marginal utility of inheritance in Chrysalis++ is not an indictment of the C++ inheritance mechanism. Chrysalis++ had few 'similar' types upon which to build.

Several weaknesses in C++ hindered the development of Chrysalis++.

- The lack of parameters to constructors for arrays of classes prevented the natural construction of process arrays. This defect applies to sequential programs as well.
- Allocation of arrays at scope entry prevents the most efficient solutions to dynamic data needs. This restriction prevents a natural and convenient dynamic allocation of child processes.
- The notion that primitive types are not classes prevents inheriting their operations. This increased the definition cost of some classes, but overall is not a major problem.
- The preprocessor based generic mechanism causes special, non-intuitive, spacing restrictions in the source code. This affects the programmers syntax in using generic classes.
- C++ operator overloading does not distinguish between the prefix and postfix forms of increment and decrement. This precludes use of a notation that C programmers find comfortable and convenient. For instance, the user cannot use these operators with atomic integers.
- The C++ handling of constructors does not allow initializing objects at known locations. This lack affected the definition of shared objects. Such a capability is outside the domain of C++.
- The lack of open classes prevents using safe C++ constructs when allocating dynamic data. This represents more of a problem in C++ than it does in C because C++ provides such clean facilities for allocating data of known size.
- C++ does not recognize `a->b` as an abbreviation for `*(a).b`. This affects the syntactic use of shared objects.
- The lack of program access to the C++ notion of type prevents insuring accurate type compatibility between processes. This too is outside the domain of C++.

The C++ programming language's compatibility with C is both a blessing and a curse. It is a blessing because it achieves wider acceptance, greater portability and an easier transition from non-object-oriented languages. It is a curse because dubious syntactic forms in C are magnified in C++ and the language is larger and more complex than its function dictates.

Creating an interface between object-oriented systems is not necessarily a straightforward task. Subtle differences in object creation, access and deletion semantics between the two systems become major inconsistencies at the interface. The object semantics for C++

and Chrysalis are both reasonable, but their combination in Chrysalis++ is less than clean. An object-oriented system for parallel programming requires careful consideration of object semantics and lifetimes.

References

[BBN, 1985a] BBN, *Butterfly Parallel Processor Overview*, BBN Laboratories, Cambridge, Massachusetts, June 1985.

[BBN, 1985b] BBN, *Chrysalis Programmer's Manual*, BBN Laboratories, Cambridge, Massachusetts, June 1985.

[BBN, 1985c] BBN, *The Uniform System Approach To Programming the Butterfly Parallel Processor*, BBN Laboratories, Cambridge, Massachusetts, October 1985.

[Crowl, 1986] Lawrence A. Crowl, "Chrysalis++," Butterfly Project Report 15, University of Rochester, Computer Science Department, December 1986.

[Hinkelman, 1986] Elizabeth A. Hinkelman, "NET: A Utility for Building Regular Process Networks on the BBN Butterfly Parallel Processor," Butterfly Project Report 5, University of Rochester, Computer Science Department, December 1986.

[Kernighan and Ritchie, 1978] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[LeBlanc *et al.*, 1986] Thomas J. LeBlanc, Neal M. Gafter, and Takahide Ohkami, "SMP: A Message-Based Programming Environment for the BBN Butterfly," Butterfly Project Report 8, University of Rochester, Computer Science Department, July 1986.

[Olson, 1986] Thomas J. Olson, "Modula-2 on the BBN Butterfly Parallel Processor," Butterfly Project Report 4, University of Rochester, Computer Science Department, January 1986.

[Scott, 1986] Michael L. Scott, "LYNX Reference Manual," Butterfly Project Report 7, University of Rochester, Computer Science Department, March 1986.

[Scott, 1987] Michael L. Scott, "Language Support for Loosely-Coupled Distributed Programs," *IEEE Transactions on Software Engineering*, SE-13(1):88–103, January 1987.

[Scott and Finkel, 1984] Michael L. Scott and Raphael A. Finkel, "A Simple Mechanism for Type Security Across Compilation Units," Technical Report 541, University of Wisconsin—Madison, Department of Computer Sciences, May 1984, A later version will appear in the IEEE Transactions on Software Engineering.

[Steinberg *et al.*, 1986] Seth Steinberg, Don Allen, Laura Bagnall, and Curtis Scott, "The Butterfly Lisp System," In *Proceedings of the 1986 AAAI*, volume 2, pages 730–734, Philidelphia, Pennsylvania, August 1986.

[Stroustrup, 1986] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

A Chrysalis++ Example

This program illustrates many features of Chrysalis++. Its core is a shared stack of integers that synchronizes access by multiple processes. This class allows instances to have different maximum sizes—it is an open class as discussed earlier.

The file "stack.h" defines the stack class. The file "stack.c" implements the stack operations that are not performed inline. The file "pusher.c" implements a process that pushes 10 values on the stack. The file "popper.c" implements a process that pops 10 values from the stack. The file "master.c" creates a stack and starts up a number of pushers and poppers.

The corresponding C code using Chrysalis takes at least twice the amount of code, and is far less readable. More importantly, the C code has more opportunity for bugs.

stack.h

```
class stack
    {
    atomic_short lock ; // true while locked for busy waiting
    int limit, top, storage[ 1 ] ;  // will allocate >1 slot

public:
    int size( int num )  // passed number of slots
        { // will execute inline
        return sizeof( lock ) + sizeof( limit ) + sizeof( top )
               + num * sizeof( int ) ; // size of storage array
        }

    void init( int num )  // passed number of slots
        { // will execute inline
        limit = num - 1 ;  // limit of indices
        top = -1 ;  // initially no entries
        lock = FALSE ;  // initially busy lock is not locked
        }

    stack( int num )  // this is the constructor
       { // will execute inline
       if ( this != 0 )
           bfCC_error( "Stack must be ‘new’d.", 0, "", 1 ) ;
       else
           {
           int nchars = ((stack *)0)->size( num ) ; // call size
           this = (stack *) new [ nchars ] char ;
           this->init( num ) ; // initialize
           }
       }

    boolean is_empty( )  // will execute inline
       { return top < 0 ; }  // zero points to first entry

    boolean is_full( )  // will execute inline
       { return top >= limit ; }  // limit points to last entry

    boolean push( element &item ) ; // these will execute out of line
    boolean pop( element &item ) ;  // and return TRUE if they fail
    } ;

declare2(shared,stack,int) ;  // parameters to size and init are ints
// macro to instantiate concrete class from generic class
```

stack.c

```
#include "chrysalis++.h"
#include "stack.h"

boolean stack::push( int &item )  // stack member function push
    { // item is a reference to an int
    while ( lock.ior( TRUE ) ) ; // busy wait while locked
        // lock.ior returns old value and sets to new value
    if ( is_full( ) )  // check stack for full
        {
        lock = FALSE ; // unlock
        return TRUE ; // failure
        }
    storage[ ++top ] = item ;  // adjust top and insert element
    lock = FALSE ; // unlock
    return FALSE ; // successful
    }

boolean stack::pop( int &item )  // stack member function pop
    { // item is a reference to an int
    while ( lock.ior( TRUE ) ) ; // busy wait while locked
    if ( is_empty( ) )  // check stack for empty
        {
        lock = FALSE ; // unlock
        return TRUE ; // failure
        }
    item = storage[ top-- ] ;  // remove element and adjust top
    lock = FALSE ; // unlock
    return FALSE ; // successful
    }
```

pusher.c

```
#include "chrysalis++.h"
#include "stack.h"

void main( int argc, char **argv )
    {
    // setup access to stack given string form of handle
    shared(stack) global_stack( handle( argv[ 1 ] ) ) ;
    for ( int i = 1 ;  i <= 10 ;  i++ )
        while ( (*global_stack).push( pnn ) ) // push the processor
            duration( 0.001 ).wait( ) ; // node number until it succeeds
    }
```

popper.c

```
#include "chrysalis++.h"
#include "stack.h"

void main( int argc, char **argv )
    {
    // setup access to stack given string form of handle
    shared(stack) global_stack( handle( argv[ 1 ] ) ) ;
    int item ;
    for ( int i = 1 ;  i <= 10 ;  i++ )
        while ( (*global_stack).pop( item ) ) // item passed by reference
            duration( 0.001 ).wait( ) ; // pop until it succeeds
        // pusher's processor number is now in the variable item
    }
```

master.c

```
#include "chrysalis++.h"
#include "stack.h"

void main( int argc, char **argv )
    {
    // node generator to spread processes around
    node_generator nodes( 0 ) ;
    // create stack for 300 elements
    shared(stack) global_stack( nodes.next( ), 300 ) ;
    // save the string form of its handle to pass to pushers and poppers
    char *argument = (char *) handle( global_stack );
    // the program's first argument is the number of pushers and poppers
    int number = atoi( argv[ 1 ] ) ;
    // start pushers and poppers.
    processes pushers( "pusher", argument, nodes, number ) ;
    processes popers( "poper", argument, nodes, number ) ;
    } // implicitly wait for pushers and popper before leaving scope
```