

CONSTRUCTING A MULTIPLE-VEHICLE, MULTIPLE-CPU SIMULATION USING OBJECT-ORIENTED C++

Michael M. Madden*, Patricia C. Glaab, Kevin Cunningham*,
P. Sean Kenney, Richard A. Leslie, David W. Geyer

Unisys Corporation
20 Research Drive
Hampton, VA 23666

Abstract

The object-oriented features of C++ simplify the design of multi-CPU, multi-vehicle simulations. Classes package data and the methods that act on the data. This packaging enables easy multiplication of objects. C++ supports inheritance and polymorphism. Polymorphism allows derived classes to redefine the methods that they inherit from their base class. Thus, client code can act on a collection of heterogeneous objects as a collection of their common base class; yet the behavior that each object exhibits is defined by its derived class type. These object features directly support the creation of heterogeneous, multi-vehicle simulations. To extend this design to multiple CPUs, developers must enable object sharing among processes or threads. Without guards, concurrent object access can lead to data corruption or program failure. This paper introduces several techniques for handling concurrent object access. Also discussed are the unique challenges to using multiple processes versus using multiple threads for multi-CPU operation. This paper uses the Langley Standard Real-Time Simulation in C++ (LaSRS++) as a successful example of applying these design techniques. LaSRS++ is an object-oriented framework for creating simulations that support multiple, heterogeneous vehicles on multiple CPUs.*

* Senior Member, AIAA

Copyright © 1998 by the authors. Published by the American Institute of Aeronautics and Astronautics, Inc. with permission.

Introduction

Popular procedural languages, such as FORTRAN77 and C, lead to complex designs for multi-vehicle, multi-CPU simulations. Procedural languages treat data and functions separately. The whole simulation program must have hard-coded knowledge of which functions act on which data and when. To reduce design complexity and retain maintainability, simulations using procedural languages frequently support a hard coded mixture of vehicles, fixed in their number and variety (if any). Our development team desired a single simulation framework capable of supporting any number and variety of vehicles. Such a simulation would boost productivity (through code reuse) and reduce development times.

The recent maturation of object-oriented languages, such as C++ and Smalltalk, offer a variety of techniques that bind function and data. These object-oriented features give developers new tools that simplify the design of complex systems. Object-oriented design (OOD) begins with defining classes. A *class* defines the attributes and behaviors common to a set of objects¹. An *object* is one instance of a class.

Classes may inherit from other classes. The *derived* class inherits the attributes and behaviors of the *base* class. Derived objects extend or specialize the capabilities of their base class. Inheritance groups classes, that share common attributes and behaviors, into hierarchies. In C++, a derived class can be assigned to a reference[†] to its base class. Clients[‡] that use the reference

[†] Unless stated otherwise, *reference* refers to both the reference and pointer types in C++.

can only access the interface of the base class. However, *polymorphism* allows the derived class to redefine the behavior of the base class interface. Polymorphism will be discussed in more detail later.

True class design does not allow clients to directly access class attributes. Clients are only allowed to interact with the object using a functional interface. These class functions are called *methods*. Methods may not represent class attributes in the same form in which they are stored[§] nor may they expose all of the internal data in a class. This aspect of object-oriented design is called *encapsulation*.

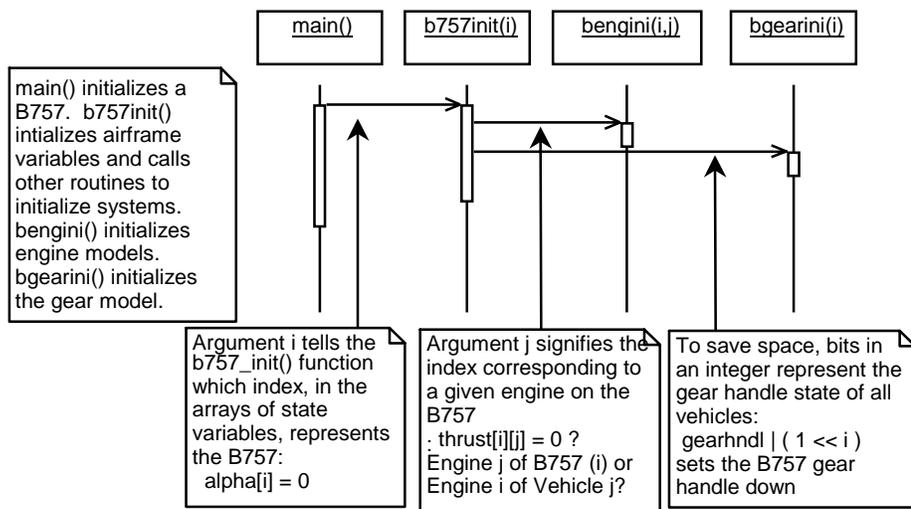
These features of objects are instrumental in designing a simulation capable of supporting any heterogeneous collection of vehicles. Object packaging (i.e., the binding of data and function in classes) and encapsulation simplify the task of creating and using multiple vehicle objects. Inheritance and polymorphism allow the same client code to operate on different collections of heterogeneous vehicles. Having established these design principles, the discussion will address the distribution of vehicles across CPUs.

In each of these sections, the discussion introduces the design concepts as they generally apply to all objects. Then it uses examples to show how these concepts apply to vehicle simulation. The examples are drawn from the design of the Langley Standard Real-time

[‡] Any object or function that operates on an object is a *client* of the object.

[§] To simplify discussion, *attribute* will refer to both its internal data and interface representations.

Initialization of an Aircraft in a Multi-Vehicle Simulation Using Procedural Programming



Initialization of an Aircraft in an Object-Oriented, Multi-Vehicle Simulation

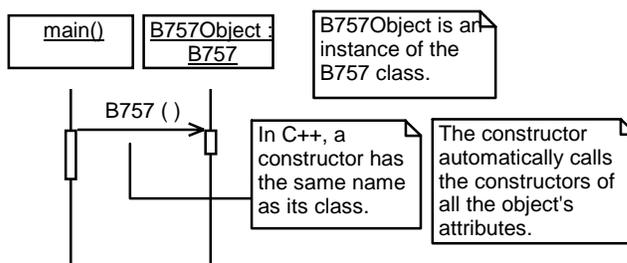


Figure 1 Multi-Vehicle Initialization

Simulation in C++ (LaSRS++). LaSRS++ is an object-oriented framework for creating simulations. LaSRS++ is currently used at NASA Langley Research Center to support its simulation facilities. An overview of the multi-vehicle, multi-CPU design of LaSRS++ is presented in the final section. LaSRS++ is framework for building a closed-loop, continuous-cyclic simulation for real-time. In other words, the simulation breaks time into equal sized frames and executes one iteration of its event loop every frame. Although this paper discusses many general design techniques, some are appropriate only for this type of simulation. When this paper uses the term “simulation” it is referring only to this type of simulation.

The Multiplicity of Objects

One advantage of classes is that they multiply easily. Once a class is developed, it takes a single line of code to create an instance. A developer can create as many objects of the class as the hardware will support. (The number is restricted by available memory. In the case

of real-time simulation, the computational speed of the hardware can also restrict the number of objects since objects must execute within the simulation's frame time). This ability demonstrates the power of encapsulation. Encapsulation allows the developer to operate on an object as a single entity even though it may internally be an aggregation of variables and other objects. Encapsulation shields the developer from dealing with the internal implementation of the object. Each object receives its own copy of class attributes. The developer does not have to manually create the desired copies of the data. For example, once a Boeing757 class has been constructed; the developer must only issue a single declaration, 'Boeing757 another_b757;', to create a Boeing757 object with all its associated data. If simulation requires another Boeing757 later, the developer simply adds another one-line declaration and another copy of data for a Boeing757 model is created.

That single line declaration does more than just create another copy of the data. It also initializes all of the data. Each class defines a special method called a *constructor*. The constructor's main purpose is to initialize all of attributes. (Constructors also create dynamically-allocated attributes.) When the constructor exists, the class has a valid initial state. In C++, constructors build on each other. A derived class automatically calls the constructor of its base class. A class constructor automatically calls all of the constructors of its attributes. This saves work for the developer.

Classes also bind data with the methods that act on them. In object-oriented programming, clients never act on object attributes directly. Instead, the class presents a functional interface to the clients. This functional interface represents all of the services available to the client. A client only changes an object's attributes indirectly, by calling a method from the object's interface. In this manner, encapsulation allows the developer to selectively expose object attributes. By design, the developer can demonstrate the minimal interface required to operate the object. This is the only interface other developers need to examine; the underlying implementation is a hidden complexity.

When operating with objects, the developer does not have to track which functions operate on a particular set

of data and when. The set of available actions is built into the class interface. Which of the class's attributes, that a particular method manipulates, has already been determined in the method's definition. The only step left for the developer is to invoke the action on a particular object. This binding saves bookkeeping at the design and construction phases. The developer is less concerned with designing function arguments that cause the function to operate on data representing a particular instance of a vehicle model. Clients also do not have to deal with the extra complexity of these function arguments, i.e. deciding which values are required to operate a given model. Objects provide a simple, consistent means of performing actions on them.

Figure 1 illustrates the differences between procedure-based and object-based simulations in implementing multiple vehicles. The figure uses a variation of the interaction diagram from the Unified Modeling Language (UML)⁷. The procedure-based simulation implements multiple vehicles using arrays (single and multi-dimensional) and packed booleans. At every level, the developer has to pass index arguments that represent the vehicle or its components (e.g. multiple engines). The developer must universally apply the same index value for a given vehicle or given component. In the engine example, the developer must know which index into a multi-dimensional array applies to an engine and which index applies to the vehicle. In the landing gear model example, the developer must also make sure that the bit position in the integer corresponds to the index position of the vehicle. In the object-based model, one constructor call initializes the object. There are no arguments that help the constructor locate the object's data. The object implicitly knows the location of its data.

Supporting Heterogeneous Vehicles

Polymorphism supports the manipulation of heterogeneous objects through a common interface. Polymorphism separates interface and behavior by allowing derived classes to change the behavior of methods declared in a base class. When a client calls a polymorphic method through a reference to the base class, the client actually invokes the behavior of the derived class. Thus, a client can act on a collection of objects through

the base class interface, yet invoke behavior particular to each object. The client class makes no assumptions about the specific objects in the collection. It views the collection only as a collection of objects of the base class. The client is only coupled to the base class's interface. One can construct more derived classes, change the contents of the collection, or alter the order of the collection without the need to rewrite the client code. In C++, developers create polymorphic methods by specifying them with the *virtual* keyword. Polymorphism is invoked when a client calls the virtual method using a reference to the base class.

In LaSRS++, the Vehicle class has a virtual method called doOperate(). The F16, Boeing757, and X31 classes redefine doOperate() to execute their unique aerodynamic, engine, and flight control models. LaSRS++ stores vehicle objects on a list of Vehicle pointers. It assigns objects of all three derived classes to the list. The event loop causes all three aircraft to fly by calling the doOperate() method for each vehicle on the Vehicle list. The event loop does not contain any pre-conceived assumption about the types of vehicles in the list or their data representation. It only sees items on the list as Vehicle objects. An F18 class could later be added to the list and the client code would remain unchanged. Figure 2 illustrates the difference in multi-vehicle operation between the procedure-based and the object-based simulation.

Polymorphism simplifies the controller component of the simulation. The controller has the same responsibilities no matter what vehicle is being simulated. The controller's responsibilities include supply services for framed execution, controlling the operational state (i.e. mode) of the simulation, and managing time. By applying polymorphism to simulation design, the developer can code the controller component so that it views all vehicle models as Vehicle objects. It interacts with the vehicles using polymorphic methods introduced in the common Vehicle base class. The controller becomes capable of operating on any heterogeneous collection of vehicle models.

Running Vehicles on Multiple CPUs

So far, this paper has discussed how object-oriented programming (OOP) can facilitate the creation of a

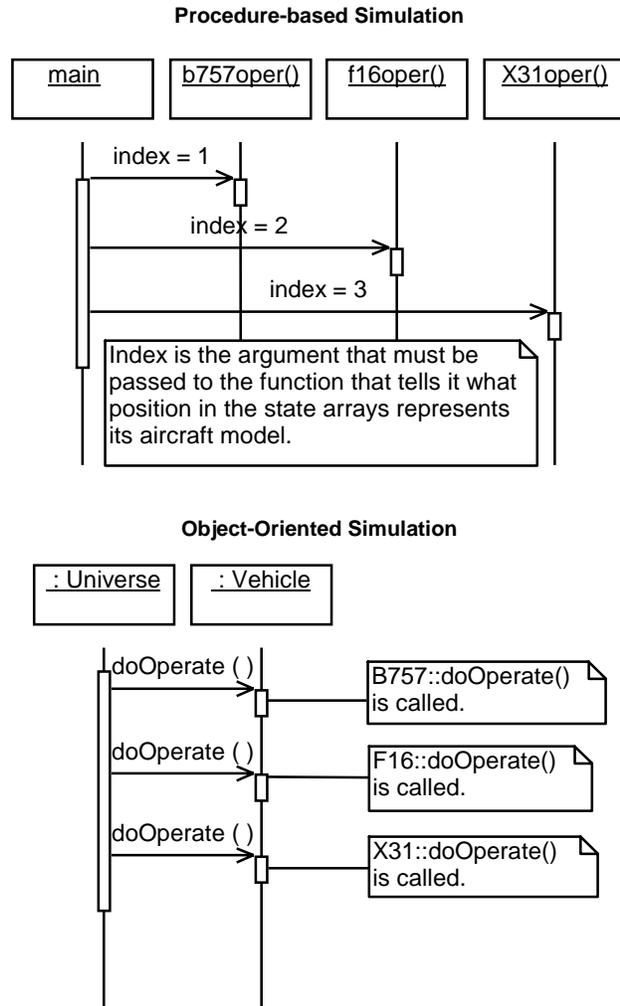


Figure 2 Polymorphism Example

simulation that operates on a heterogeneous collection of vehicles. Without any more design considerations, this simulation is guaranteed to work only on a single CPU. Partitioning vehicles across multiple CPUs requires more planning. This discussion addresses multiple-CPU operation on a single computer. Simulations, which are distributed over multiple-computers, have other challenges, which are beyond the scope of this paper. This discussion also does not provide a complete review of all the dangers inherent in a multi-CPU programming. It focuses on problems unique to object-oriented systems.

There are two methods of running vehicles on multiple CPU's: multiple threads or multiple processes. The main difference between the two methods is memory access. Threads spawned by the same process share a

common address space. Each process has its own address space, inaccessible to other processes. A common address space makes data easily sharable among threads; all that is required is a pointer or variable reference to the data. Processes, however, must transfer data through an inter-process communication (IPC) mechanism such as shared memory. Shared memory provides a means of creating a common address space among processes. Shared memory enables the developer to design a multi-process simulation similar to a multi-threaded simulation. With a little extra work, the simulation can be made portable between systems that support threads and those that do not. Because of these compelling advantages, this paper will only discuss a multi-process design that uses shared memory.

The first subsection, “Concurrent Access of Shared Objects”, discusses the challenges that the two multi-CPU designs share in accessing shared objects. The two multi-CPU designs also have unique problems in sharing objects related to their memory models. These differences are the topics of the subsections “Multiple Processes” and “Multiple Threads”.

Concurrent Access to Shared Objects[¶]

An object-oriented simulation designed for multi-CPU operation has a set of shared objects. More than one process operates on a *shared object*. These objects can represent frame rate, time, operational mode, world characteristics (radius, gravitational model, atmospheric model), and even the vehicle models (in combat simulations or simulations with collision avoidance). Some of these objects offer services required by all simulation models. Some of them also require that only one process invoke operations that propagate their state from one frame to the next. Whether using multiple processes with shared memory or multiple threads, restrictions must be imposed on shared objects. Otherwise, the simulation will act unpredictably or even crash

In OOP, all objects are accessed through their methods. Class methods are either mutators or constant. A mutator method modifies one or more object attributes. A constant method does not modify any object attributes.

[¶] For the sake of brevity, this section uses the term “process” to refer to both processes and threads.

In a multi-process environment, the developer must guard against concurrent use of a mutator method with any other method. When mutator methods are concurrently invoked, they may attempt to modify the same attributes. The conflicting computations will place the object in an invalid state. When a mutator and a constant method are simultaneously active, the constant method may return an intermediate value. Both situations cause erroneous behavior or program failures.

The first task of the simulation developer is to identify the mutator and constant functions. C++ supports this division of methods. Developers can declare a method constant by tagging it with the *const* qualifier. C++ enforces the qualifier. It is a syntax error for a *const* method to modify class attributes either directly or indirectly (e.g., by calling a mutator method)[#]. The developer must then uncover those situations in which a mutator and other methods may be invoked concurrently. Before using one of the techniques described above, the developer should first examine the mutator method for possible conversion into a constant method. Developers have a tendency to make class attributes out of all the variables used by class methods. Some of these “attributes” can be converted to stack variables (i.e., local variables, method arguments, or return values). If all of the “attributes” modified by the method can be converted into stack variables; then the developer can qualify the class as constant. This solves the concurrent access problem since concurrent execution of constant methods is always safe. Otherwise, this situation must be eliminated by using one of the techniques described below.

Developers can use several techniques to guard against concurrent use of mutators with other methods. Access to the class can be guarded using a mutual exclusion

[#] In C++, a class attribute can be qualified with the *mutable* keyword. Mutable attributes can be changed in a *const* method. The authors recommend that developers avoid the use of the *mutable* keyword for this reason. Not all compilers are equally adept at identifying situations where class attributes can be indirectly modified, particularly when data is aliased by a pointer.

mechanism (mutex^{**}). The mutex becomes an attribute of the class. Methods must first acquire the mutex before performing their function. This method guarantees that class methods will not be called concurrently. This technique is ideal for situations where client requests must be acknowledged immediately. For example, LaSRS++ contains a SharedMemory class that allows the dynamic allocation of blocks of memory. The SharedMemory class contains a semaphore. When a client calls the allocate() method, the method first attempts to acquire the semaphore before reserving a memory block of the requested size. The semaphore prevents two processes from reserving shared memory space simultaneously which can lead to overlapping blocks and corruption of shared data.

A second technique buffers attribute changes. Other processes modify the buffer and do not directly interact with class attributes. The buffer can act on behalf of one or more classes. The classes copy data in the buffer when they are ready to update their state. The buffer still requires a mutex to guarantee that its contents are not modified while the classes copy the data. However, the buffer consolidates multiple class-level synchronization actions into a much smaller number actions on the buffer. This option works well in situations where the effect of the attribute change can be deferred. Thus, this option is frequently used for user interfaces or hardware communication classes. In LaSRS++, the X-based interface changes the attributes of specific aircraft models using a buffer in shared memory. Before the aircraft updates its state, it examines the buffer for changes and copies new inputs into the appropriate attributes.

A third technique partitions program execution into blocks of events. In one event block, processes update objects they created; no interprocess communication is done. In the next block, processes communicate with each other using only constant class methods. For example, in LaSRS++, the vehicle models are executed as an event block before the relative geometry between

^{**} Some operating systems offer a synchronization mechanism called a mutex. Mutex is used in this paper to denote any mutual exclusion mechanism including semaphores.

them is calculated in the next event block. Since each process could exit the event block at a different time, the processes must communicate to each other when they are finished. This can be done with counting semaphores [1], condition variables [2], or a set of booleans (a.k.a. flags) in shared memory.

Shared flags are the simplest mechanism to implement and will be used as the example. LaSRS++ uses this technique to signal when all processes have updated their vehicles. One flag is created for each process. The flags are all initialized to false; this indicates that the processes have not updated their vehicles. As each process completes the event block, it sets its associated flag to true. Then, the process enters a *while* loop that exits only when all the flags are true.

The danger of this design is in resetting the flags. The flags cannot be reset until all processes have exited the while loop, and the flags must be reset before a process returns to the event block. Closed loop, continuous cyclic simulations have an advantage in this situation. They guarantee the following: processes synchronize at the start of frame, processes cannot overrun their frames, processes have dedicated CPUs, and processes cannot be interrupted during normal operation. These characteristics create opportunities where the flags can safely be reset. In LaSRS++, one process, designated the “main” process, resets all the flags before the frame ends. This operation is considered safe because the main process performs many functions after the vehicle update that the other “auxiliary” processes do not. Thus, the main process is always the last process to complete its event loop.

Division of the program into event blocks is the preferred method. It requires the least amount of work since it introduces synchronization at the program level. Synchronization does not have to be designed at the object-level or in association with buffer copies (i.e., data exchanges). Synchronization is associated only with events. Within these events, any concurrent access of objects is designed to be safe.

Unsafe concurrent access to objects is not the only factor that can compromise data integrity. Code optimization can cause similar problems. Optimizers attempt to keep as much data in the CPU’s registers as possible.

Optimizations may keep the value of a shared object's attribute in a register. While the attribute is in the register, the process running on that CPU does not recognize changes made to the attribute by other processes. This situation defeats careful planning to avoid unsafe concurrent access of mutator methods with constant methods. A mutator called in one process changes the attribute's value; but the constant method called immediately afterward by another process continues to use the old value since it already exists in a register. Fortunately, C++ provides the type qualifier *volatile* to tell the optimizer that a variable may change in unpredictable ways and, thus, must always be read from memory². All class attributes used across processes must be declared with the *volatile* qualifier.

Multiple Processes

Multiple processes share objects by placing the objects in shared memory. Using shared memory to create a common address space for the processes requires a fair amount of design work. Fortunately, C++ features seamless creation of custom allocators^{††}. For each class, developers can redefine the *new* operator^{‡‡}. The custom *new* operator is also inherited. If introduced in a base class, all classes in the hierarchy will use the custom new operator for dynamic allocation unless they redefine it. Shared classes can redefine their *new* operator to place them in shared memory. By redefining *new*, shared classes and non-shared classes use the same standard C++ syntax for dynamic allocation. A shared class can later revert to a non-shared class by commenting out the redefined *new* operator.

Redefinition of the new operator is almost required for custom placement of objects. Object constructors cannot be called explicitly except to create temporary objects resulting from a type conversion³. The *new* operator is the only means to invoke a class constructor on a dynamically allocated object. However, redefining

^{††} An allocator reserves memory space for a dynamically created object. In C++, the operator *new* performs dynamic allocation of data.

^{‡‡} The *delete* operator must also be customized. The delete operator reclaims memory space. For brevity, the *delete* operator is left out of the discussion.

new for a class causes all objects of that class to be custom allocated. This can only be overridden by explicitly invoking the global *new* operator using a scope qualifier, i.e. *::new*. If only one or two objects of a class need to be shared, C++ provides a variation of the *new* syntax called "placement syntax"³. Placement syntax allows the developer to specify the address where the object will be constructed. Sufficient memory to contain the object must already be reserved at that address when the "placement new" is called.

Customizing the *new* operator for shared memory allocation requires a memory manager. The memory manager creates the shared memory segment and allocates space on demand. It may reuse memory reclaimed from deleted objects. Since multiple processes could attempt to allocate or delete objects simultaneously, the memory manager must guard its allocation and deletion functions with a mutex. The developer usually has to create the memory manager; pre-packaged solutions may not be available.

Since an object can contain other objects, shared objects can dynamically create other shared objects. These shared objects will then contain pointers to other locations in the shared memory segment. These pointers are valid for all processes only if all processes attach the shared memory segment using the same starting address. Fortunately, UNIX allows the developer to specify the starting address of the shared memory segment. However, the authors' experience has shown that, on some UNIX variants, it is difficult to select a valid starting address. UNIX variants may reserve ranges of addresses for specific uses (with scant documentation on this topic). The address for the shared memory segment must not collide with these reserved spaces or addresses assigned to the program's text segment or stack; if it does, the operating system will reject the attach request. Sometimes, the developer must resort to compiling a version that lets the OS pick the starting address and prints this address. Then the developer can reconfigure the simulation to use this OS-picked address.

Shared objects that reference other shared objects are unavoidable in a flexible, configurable simulation. In fact, relying on object relationships is the most efficient

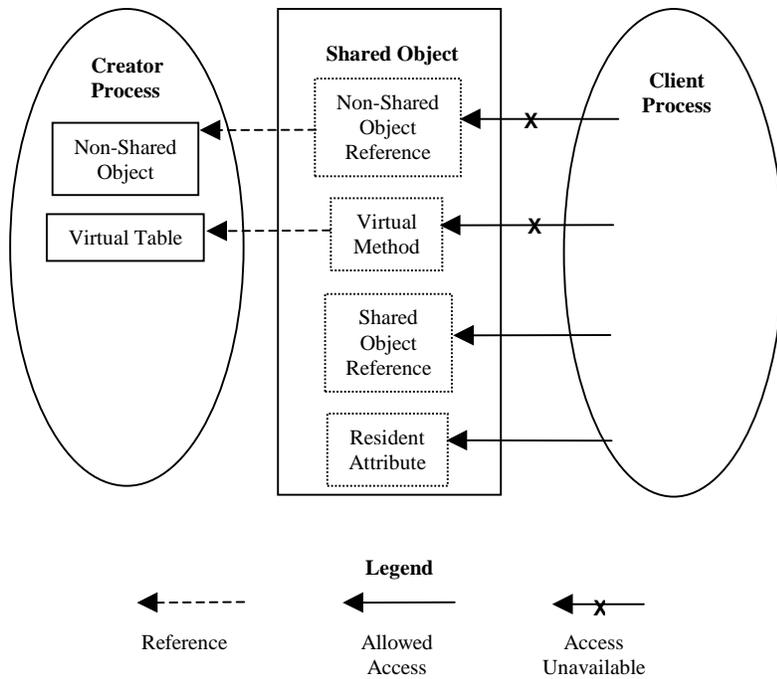


Figure 3 Access to Shared Object in Shared Memory

means for processes to find shared objects. For example, a list of aircraft in the simulation is placed in shared memory. Processes use the list to find the location of each aircraft. The processes could then use the aircraft to find its landing gear model, etc. This method does not significantly differ from the way in which processes find their non-shared objects. The main difference is that the processes need a map to locate the first objects in these relationship chains. This map must exist at a fixed address in shared memory. These maps should contain pointers to very few items; the design should rely on object relationships as much as possible.

Shared objects can also reference non-shared objects. These references are only valid for the process that created the non-shared object. If another process attempts to access the non-shared object, a segmentation violation will result. Developers must ensure that all the objects they intend to share across processes are placed in shared memory.

However, there is one non-shared attribute that developers have no control over. Objects with virtual methods have a hidden pointer to a virtual method table. This prevents processes from calling polymorphic methods on shared objects, which the process did not create. C++ implements virtual methods using a virtual

method table (vtable). The vtable contains a list of function pointers. Each entry in the list is associated with a virtual method. The pointer in the entry refers to the actual implementation that will be run when the virtual method is called on an instance of a particular class. When a class redefines a virtual method, the entry for that virtual method in the class's vtable is assigned a pointer to the class's implementation. Since objects of the same class would have identical vtables, most compilers attempt to create one vtable for all instances of the class. The vtable is a global object in the process's address space. The class obtains a hidden attribute, a pointer to its vtable. When a virtual method is invoked, the program references the vtable using the vtable pointer. Then, it looks up the appropriate

method to call in the vtable and executes it. A shared object with virtual methods references a vtable in the process that created it. Other processes attempting to call a virtual method on this object will experience a segmentation violation.

Even if the developer finds a way to place the vtable in shared memory, most UNIX variants do not support the placement of functions in shared memory. Thus, the pointers in the vtable can only reference addresses in the creator's address space. The inability to invoke virtual methods on a shared object severely restricts the abstraction of shared objects. Clients must know the exact class from which they are requesting services. For example, a simulation may be required to support multiple atmospheric models. The developers initial inclination is to design a base class called Atmosphere with a virtual method called calculateProperties(). Each model is represented by a derived class that redefines the calculateProperties() method. However, all processes must share this class since the atmospheric properties are an input to all the vehicle models. Thus, the designer must bundle all of the atmosphere models into one class and provide non-virtual methods to execute each model.

The process that created the shared-object can still execute the object's virtual methods, however. In LaSRS++, each process keeps track of which vehicles it created. Those processes are responsible for calling the virtual methods, such as the `doOperate()` method introduced earlier.

All processes can still execute the non-virtual methods of a shared object. In this case, the separate address space presents an advantage. If two processes call the same method on two different objects, two different copies of the method are executed. There are no situations where the same copy of a method will be called concurrently. The developer does not need to be concerned with designing functions that are re-entrant.

Multiple Threads

Sharing objects in a multi-threaded environment is simple. All objects and all threads live in the same address space. Any thread has access to any object. The multiple thread design eliminates many of the restrictions found in the multiple process design. Developers do not need to create custom allocators. All object references are valid no matter which thread created the referenced object. Any thread can call a virtual method on a shared object that it did not create. In this sense, multi-threaded design is better suited to object-oriented programs.

However, the common address space does present a new challenge. Not only do multiple threads share data; they also share functions. When two threads invoke the same method on two instances of the same class, both are operating the same copy of the method. This situation is not safe unless the method was designed to be re-entrant.

Re-entrant methods do not intentionally make possible the concurrent modification of the same object. Since each thread receives its own stack, local variables and method arguments are guaranteed to be safe from concurrent modification. (The reference in a pass-by-reference argument is the data actually stored on the stack and is safe from concurrent modification. The referenced object is not protected.) All attempts should be made to localize variables used by the method. (Local variables that are declared *static* do not exist on the

stack. They are a form of global data.) Re-entrant methods avoid direct modification of global or heap data (i.e., non-local data). (If non-local data that the method directly references is constant or guarded by a mutex, the method is essentially re-entrant.) Access to non-local data should be done through the method's argument list. If the argument list contains mutable pass-by-reference arguments, client code can pass the same object to multiple, concurrent invocations of a method, causing the same object to be modified. However, this does not invalidate a method as re-entrant. A re-entrant method only guarantees that the method itself will not modify the same object concurrently, outside of the control of the client. A re-entrant method does not prevent clients from causing concurrent modification of objects.

Good object oriented design leads to re-entrant methods. A well-designed object encapsulates its data, allowing access only through its functional interface^{§§}. The methods in a class should only interact directly with attributes of the class or with local variables. The need to obtain information from outside of the class should be rare. When needed, external data should not be obtained through global access. Outside information should be passed through method arguments. If possible, arguments should be pass-by-value. Pass-by-value makes a local copy of the argument; modifying the copy does not modify the argument. However, pass-by-value is computationally efficient only for intrinsic types and very small objects. Larger objects are usually passed by reference. Pass-by-reference introduces an alias that could lead to concurrent data modification. If developer does not intend that the method modify the referenced data, the developer can enforce this design by declaring that the reference points to a constant object. The method cannot directly modify constant objects or call their mutator methods. The compiler will issue a syntax error for any attempt to modify a constant object.

^{§§} In C++, a class can expose its encapsulated data to another class or function through use of the *friend* keyword. Friendship breaks encapsulation and should not appear in objects intended for multi-threaded applications.

Classes that meet the above definition have a high level of cohesion.

C++ implements method access to class attributes in a thread-safe manner. Methods access class attributes through a hidden argument, a pointer to the class. The pointer is named *this*. When a method is concurrently executed on different objects by different threads, each invocation uses a different *this* pointer from the thread's stack and thus modifies a different object. However, a method is no longer re-entrant if it modifies a static attribute that is not guarded by a mutex. Static attributes are a form of global data; the attribute is shared among all instances of a class. Classes should avoid mutable attributes that are static. All static data should be constant.

Sometimes there are compelling reasons to use global data. The developer can use object-oriented techniques to easily identify and guard this data. Any mutable global data should be placed in a class utility. Class utilities are classes where all of the data and methods are static. The data should be encapsulated and accessible only through methods. Methods can guard the global data with a mutex, relieving clients of this responsibility. Static methods must be invoked using a scope qualifier. The scope qualifier identifies that the method call may access global data, helping developers identify client methods that may not be re-entrant^{¶¶}.

Conclusions

The object-oriented features of C++ simplify the construction of a simulation that supports multiple, heterogeneous vehicles. Classes can aggregate data and constructors. With a single line of code, an object of a class can be created with a full, initialized copy of all its necessary data. Classes bind functions to data. This binding provides a single, consistent mechanism for calling actions on class attributes. The developer is freed from designing how functions will manipulate the data representing particular copies of a vehicle model. Through polymorphism, clients can manipulate a heterogeneous collection of objects as if they were objects

^{¶¶} There are other situations where a scope qualifier is used with a method call. The point here is that static methods can only be called using a scope qualifier.

of a common base class. Yet, the clients will invoke behavior specific to the actual class of the object. Polymorphism allows developers to declare a method in a base class that derived objects can redefine, essentially separating interface and behavior.

Applying these object-oriented techniques to a multi-CPU environment requires further consideration. The developer must be cautious not to modify the same data concurrently and to make sure that all processes view changes in data. The latter is accomplished by using the *volatile* qualifier for all data that can be read by a different process than the one that modifies it. There are many ways in which the former can occur. The developer can limit the possible situations by frequent use of the *const* qualifier on methods and pass-by-reference arguments. Since the compiler enforces the *const*, the incidents of accidental changes to data, concurrent or otherwise are reduced. Applying *const* also helps developers identify mutator methods and, thus, situations where unsafe concurrent execution of methods may exist. To increase the number of constant and re-entrant methods, developers should also move as much data as reasonable to pass-by-value arguments and local variables. Developers should avoid the use of global data, particularly static class attributes and static local variables. All other global data should be placed in class utilities where access to them can be controlled, including the use of mutexes to guard access. The use of global data encapsulated in a class utility is also more easily identifiable.

In a multi-CPU environment, concurrent access of a mutator method with other methods can lead to erroneous behavior and program failures. Developers must identify and prevent these situations. Mutexes can guard against concurrent access of class methods. Buffers can defer object attribute changes; the changes then occur at once under controlled conditions. Dividing the event loops into blocks of activity can separate large-scale use of mutator methods by individual processes from concurrent use of constant methods.

Two multi-CPU environments are best suited for object-oriented programs: threads and multiple processes using shared memory. Threads are conceptually the simplest to design for. However, all code and data are shared in

a multi-threaded environment. This increases the possibility of unsafe concurrent access to data. It also introduces the problem entering the same copy of the method code concurrently. Unless the method was designed to be re-entrant, this concurrent execution of the method could lead to concurrent modification of data, particularly global data. Using multiple processes with shared memory requires more work. Developers must create custom allocators that place and initialize objects in shared memory. Shared memory must be attached at the same address for each process in order to maintain the validity of references to shared objects. Shared objects can reference non-shared objects. These references are only valid in the process that created the non-shared object; attempts by another process to access them will lead to program failure. Because of the manner in which virtual methods are implemented, processes cannot execute virtual methods on shared objects that they did not create. Despite these restrictions, the multiple process design has some distinct advantages. The developer has control over which objects are shared. Each process has its own copy of class methods; this design does not have the same problem of method re-entrance that the multi-thread design has.

LaSRS++ exemplifies these techniques. It is a successful implementation of an object-oriented simulation that supports multiple, heterogeneous vehicles running over multiple CPUs.

Bibliography

¹ Booch, Grady. *Object-Oriented Analysis and Design With Applications*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994. ISBN 0-8053-5340-2.

² Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, 1997. ISBN 0-201-88954-4.

³ *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*. ANSI X3J16/96-0225, 1996.

⁴ Robbins, Kay A., Robbins, Steven. *Practical UNIX Programming: A Guide to Concurrency, Communication, and Multi-threading*. Prentice Hall, Upper Saddle River, NJ, 1996. ISBN 0-13-443706-3.

⁵ Gallmeister, Bill O. *POSIX.4: Programming for the Real World*. O'Reilly & Associates, Inc., Sebastopol, CA, 1995. ISBN 1-56592-074-0.

⁶ Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Publishing Company, Reading, MA, 1992. ISBN 0-201-56364-9.

⁷ Quatrani, Terry. *Visual Modeling With Rational Rose and UML*. Addison-Wesley, Reading, MA, 1998. ISBN 0-201-31016-3.