subject: **C++/P: An Environment for Real-Time Embedded System Development**
Charge Case 116610-1000
File Case 40548-100

date: **May 15, 1984**

from: **J. O. Coplien**
IHP 55437
1E233 x35384
55235-840515-02TM

TECHNICAL MEMORANDUM

1. INTRODUCTION

The new circumstances under which we are placed call for new words, new phrases, and for the transfer of old words to new **objects***.*

— Thomas Jefferson

Object-based system design is a technology based on data abstraction and the allocation of knowledge and expertise to system modules. Object technology has been used as the basis for many recent systems and programming languages such as SmallTalk-80,™ CLOS, and **C++**, though the concepts of object-oriented programming have been around much longer in Simula67[1] and similar languages. The object paradigm has potential for increasing software productivity through architectural support, powerful packaging constructs that underly code reuse, and improved coding expressiveness.

Our applied research in switching software technology is exploring the merits of object technology for improving the lot of software development in general, and of embedded system software development in particular. Embedded systems, such as stored-program-control switching systems, have stringent real-time requirements and are generally large (on the order of hundreds of megabytes of software) and complex. Their Software Development Environments have special needs because these systems are large, they are real-time, they are so reliable as to be continuous running, and their service life can span several decades. The systems need to be evolved, to be featured to meet customer needs, without interrupting service to end customers. We feel the object paradigm will be especially useful for formulating and evolving architectures for such systems, for their fault recovery, and for their software maintenance.

To help deal with the challenges of embedded system software development, our projects have adopted many of the advantages of object programming as offered by the **C++** language. To this base we have added functionality to incorporate some powerful features of the object paradigm that are peculiar to some of the other environments mentioned above. But first and foremost, we started with a switching system model based on fundamental object concepts and asked what objects meant in a real-time, continuous running but dynamic environment. This model emphasized some important attributes of the object paradigm, to which we added a few of our own:

- *The importance of object firewalls:* Objects as software development units can be used as self-contained building blocks with well defined interfaces to the outside world. Not only does this help the system architect in designing and maintaining system modularity, but also can give programmers well defined development domains. Developers can introduce their own objects into a system without being able to corrupt other objects, so the continuous running requirements of the system can be enforced.

- *Object Member Functions as Units of Synchronization:* Task synchronization is an important concept in real-time systems. Object *member functions* (procedures or operations) can be treated as the indivisible operations that are available to the programmer, much as the ‘‘read-modify-write’’ machine language instruction was an indivisible operation available to the programmer of decades gone by. The primitives supported by objects can be offered at a higher level, though: for example, ‘‘create virtual phone port’’ may now be a primitive operation on a type like ‘‘telephone call.’’ If users create their own primitive operations, the underlying object machine can enforce time limits on the the execution of those primitives so that real-time requirements and system sanity are preserved.
- *Objects in a Distributed Environment:* If objects are self-contained development units, then it seems natural to make them the units of partitioning in a distributed processing environment. Objects should be able to be thought of not as parts of a larger program, but as self-contained and largely autonomous entities. Communicating objects should not have to know whether or not they are in the same ‘‘program’’ or on the same processor.

Support for these fundamental concepts was realized through the system architecture, through hardware and, last but not least, through compiler support. The **C++/P** compiler was created to give system developers an environment where object protection and object communication could be dealt with at the language level. This new compiler was used in several prototype projects in AT&T Bell Laboratories.

2. TASK OBJECTS

Objects, like people, embody a certain realm of expertise. Like people, they communicate to each other based on the expertise they have and on communications that were carried on previously. Objects communicate with each other either by examining public state variables or by invoking member functions of other objects.

Our prototype switching system architectures use objects to achieve synchronization, performance and security goals. These systems are distributed systems: this makes it possible to easily put up *real* hardware firewalls (between processors) and to grow the system (by adding new processors) to meet performance demands. Most objects’ member functions execute as run-to-completion tasks in some of our prototypes: communication between objects causes a message to be queued to the receiving object, but the sender continues to execute to completion. Objects exhibiting this behavior in their operations are called *task objects*. Direct access of data between task objects is disallowed[1]. The execution of a member function never blocks, but must run to completion within a pre-specified amount of time. This means, for example, that object member functions never return values directly (because they don’t return to the caller directly) and are always of type **void**. (The receiving function may of course transmit information to the sender in a return message at a later time). This makes it easy to analyze and enforce system real-time constraints, and greatly simplifies maintenance and recovery procedures.

This architecture has a significant impact on the ways that objects interact, and both the philosophy and mechanics of communication between task objects are different than those of regular **C++** objects. (Some objects still behave as canonical **C++** objects are supposed to). **C++/P** is designed to be integrated into the run-time environment in the way it supports object communication. In **C++**, the invocation of an object member function is implemented in the same way as a C function call where flow of control is passed to the called routine (and usually back again). **C++/P** automatically transforms object operations into messages (see Appendix 1) that are dispatched to the receiving object by the ‘‘virtual object machine’’ (see Figure

---

1. One can argue against using public data as a mechanism for interfacing between objects. Making data public leaves it open to corruption—either accidental or malicious—by other objects on the same processor. It is difficult to implement inter-object data access if the party objects are in different address spaces (as they may be for protection, or because they are on separate processors in a distributed environment). Most data is kept in objects’ private areas when doing good object design. The only ‘‘doors’’ in object firewalls are then through member functions that can validate their arguments against the current object state.

2-1)[2]. These messages contain:

— An identifier for the destination object;
— The name of the destination object's type;
— The name of the member function being called in the destination object;
— The arguments to the member function, packaged into a machine-independent string;
— A format string describing how to decode the argument string, and;
— An indication of the protection domain environment of the originator (see the next section).

The message causes the appropriate member function of the specified object to be invoked with the arguments provided by the originator. The actual call may not take place for some time after the message was sent, however; this is up to dispatching and scheduling policies held by the virtual object machine.

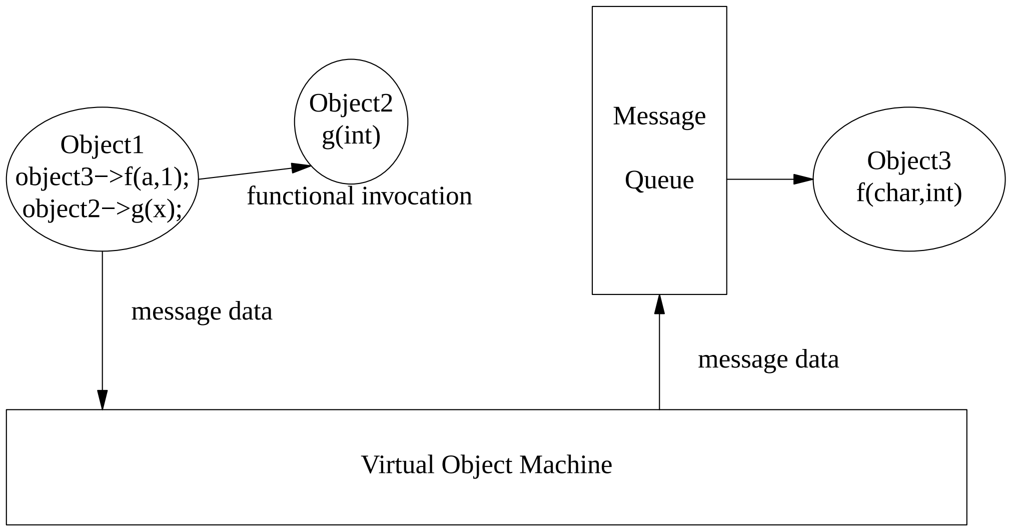


Figure 2-1. OBJECT COMMUNICATIONS IN C++/P

One interesting aspect of passing arguments between tasks is the treatment of pointers. The argument decoding string contains a special notation for pointer arguments so they can be identified and treated appropriately by the receiver. Otherwise, pointers are dereferenced one level and the value of the item pointed to is sent in the inter-object message. Tracing pointers stops after one level: if the pointer points to another pointer, or to an aggregate that contains a pointer, the values of the second-order pointers are passed as NIL. Not doing this could result in very large argument strings that might result from tracing down all pointers in a complex data structure, as might happen if the element of a tree structure or long linked list were used as a task object function argument. This also sidesteps the problem of detecting circularly linked structures.

This implementation of inter-task messages has several advantages over the use of explicit message primitives. The user can program using standard **C++** syntax, and the semantics of member function invocation are unchanged except for the asynchronous nature of their execution. There is no need to maintain separate administrative structures that define message interfaces between cooperating objects; the compiler automatically generates the message formats from member function argument declarations[3]. Furthermore, there is no need to build byte swabbing tables to rearrange the alignment and layout of

2. C++/P programmers are given the illusion that they are programming an object-based machine. This illusion is provided by the compiler and by an underlying operating system called the *object manager*. It is the object manager that dispatches messages.

3. It is of course assumed that the sender and receiver both understand the same declaration of the receiver member function. In C++ this is usually done by placing object type declarations in header files; these header files are *#include*d by both the user and provider of an object interface.

primitive types when messages are sent between unlike processors. Messages are sent as machine-independent ASCII strings. The virtual object machine need only understand the alignment and size properties of primitive types on its local machine, and use that information along with the argument decoding string (also generated automatically by the compiler) to change the machine-independent data representations into internal form. All the details of message generation, ASCII formating, pointer following, *etc*. are handled automatically by the compiler and are invisible to the user.

3. RUN-TIME PROTECTION

One aspect of our local object work was to look at the benefit of run-time protection in maintaining the security of continuous-running systems. The **C++/P** compiler supports the concept of *protection domains*, a run-time mechanism used to implement the ‘‘firewall’’ properties of object-based design.

3.1 Object Interactions, Scoping and Protection

It was mentioned above that objects are like people in the ways they communicate with each other. The kinds of things that they tell the outside world, along with the actions they can be expected to carry out in response to a request, are the *public parts* of the object.

But objects also maintain private information that is never made directly available to the outside world. This local information is used to store data (knowledge) about the object’s realm of expertise or responsibility. It usually wants to protect those data from corruption by code outside its firewalls. Outsiders are usually prevented from reading an object’s private information, first of all, because they probably wouldn’t know what to do with it if they could read it, and secondly, because the object may actually have a good reason to keep the information private (for example: lists of computer passwords or personnel data). But objects carry on their day-to-day lives and chat with each other without revealing the inner secrets of their private parts.

It is sometimes the case, however, that a pair of objects will find themselves in roles (areas of expertise or responsibility) where it is mutually beneficial for the objects to share their private information just between themselves. For example, an I/O front end system may have separate objects to handle a keyboard and a screen. The two objects should be tightly coupled because they share hardware drivers or device state information, but they are separate objects that a user may wish to treat separately. Furthermore, users should not be able to directly access the device state information associated with the terminal. The **C++** language provides a mechanism for objects to declare *friends* that have access to their inner secrets. This changes the nature of the contract between the objects, but objects that are not friends are prevented from accessing the ‘‘internals’’ of either of the mutual friends. For example, if a computing system was set up so that every programmer had a *programmer* object, programmers may want to make the system *mail* object a friend of *programmer* so it can deposit mail in their object:

```
class programmer {
friend mailroom;
    char * mailbox;
public:
    . . .
} mary;
class mailroom {
    . . .
    void sendmail(programmer whom, char * message)
    {
        . . .
        whom->mailbox = message;
    };
} mail;

    . . .
    mail.sendmail(mary, "hello, mary");
    . . .
```

The *friends* mechanism can be thought of as an extension of the **C** scoping rules, and it is enforced at compile time. Compile time is the time at which object ‘‘contracts’’ are drawn up: the compiler can see the ways in which the objects will interact and make decisions about whether the interactions are legal. If an object tries to access the private data of another object, the compiler can generate a compile-time error and terminate object code generation. The objects in a processing community form a ‘‘social contract’’ of sorts at compile time. These contracts between object types are defined by their interfaces, by what parts are public and private, and by which objects access the public parts (and, for friends, private parts) of others. The definition of an object type with its interfaces and visibility attributes is called a *class*, a template used to create all objects of that type. It is these class definitions that are used by the compiler to police object interactions. It is assumed that all the objects are playing by the rules—and the compiler, being the enforcer of the rules, can boot anyone out who is violating someone else’s space.

But compile time checking turns out to be a weak mechanism for controlling access in a secure system. An outsider can make a copy of an object type’s compile-time declaration (class), rearrange the `public` attribute labels and then recompile their own program. That user can gain access to any of the private elements of any object just by getting hold of its class definition. This is not a serious issue in a system where all the objects are assumed to cooperate (as is certainly the case when one programmer writes the entire program). But if a system is to be customized and augmented by programmers other than the provider, then special precautions must be taken if the private elements of the provider’s code are to be protected. And compile time scoping is inadequate for this purpose.

This problem can be solved by enforcing protection at run time. Protection information can be put in every object’s machine code so a malicious programmer cannot usurp protection mechanisms by playing games with scoping mechanisms at compile time.[4] Furthermore, deferring protection domain analysis until run time makes it possible to dynamically reconfigure object access permissions while the system is running. This will be useful for changing the visibility of object member functions and data as they go through various stages of initialization or recovery.

4. Another mechanism for doing this would be to have the compiler own all the object type interfaces (class definitions) for the entire system. All system object type interfaces would be declared in a centralized repository that the compiler would use to specify object layouts and enforce protection. However, complex questions arise: can a programmer then declare a class of their own, inside their own program, for their own use? If so, why can they not declare their own version of one of the types ‘‘owned’’ by the compiler, and use their own type to usurp protection mechanisms instead of defaulting to the compiler’s version? Solution of these problems at compile time involves complex semantic analysis beyond the scope of most current compiler technology.

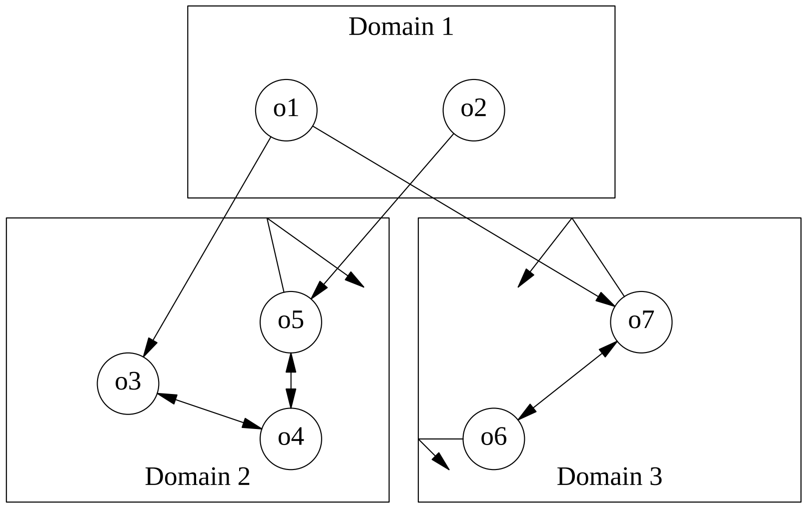
3.2 Protection Domains

C++/P models the protection environment as being made up of multiple *protection domains* in which objects run. These domains correspond to different levels of strength, or to different classes of programming power. The ‘‘virtual object machine’’ defines how domains can operate on each other. A simple example of protection domains is the division between kernel and user space used in many common minicomputers. User programs can access code in the user domain, and kernel code can access everything. The **C++** protection domains are much more general than the kernel/user model, however, and have the following distinct and noteworthy advantages:

- They are dynamic and can change at run time;
- There can be any number of objects in a domain at any given time, from zero to arbitrarily many;
- Protection domain granularity can be as large as many objects or as small as an object member;
- There can be virtually any number of protection domains; and
- The domain protection model can be non-hierarchical in nature.

Domains can be thought of as independent communities of mutually interacting objects. The protection domain model is reminiscent of the *friend* structure of **C++**. Like *friends*, the protection domain model supports a ‘‘clique’’ structure where objects in the clique define each other as all being friends. And as with *friends*, protection domain relationships between objects can be non-reciprocal: an object may gain access to another’s operations without the other object being able to operate on it. However, protection domains make the clique a visible administrative unit, whereas in *friends* the existence of a clique has to be inferred from a collection of individually declared friendships. Moreover, protection domains are bound at run time under the control of the virtual object machine: this makes them more flexible, and also thwarts attempts to undermine the protection mechanism by compiling against adulterated versions of source interface definitions. Objects in any given protection domain are usually owned and administered by cooperative (*i.e.* mutually non-malicious) programmers. These programmers’ objects may not be *friends* in the **C++** sense so that they can protect themselves against accidental contamination by neighbors. Subject to scoping (public/private/friend) the objects in a domain can access each others’ data.

An object can receive protection domain attributes either by inheriting them from the protection attributes of its defining class, or by adopting explicit attributes given to the object itself when it is instantiated. Note that protection domains differ from *friends* in that different objects of the same object type (class) may be in different domains, whereas all objects of a given class share the *friend* attributes given them at compile time.



This picture shows how objects might interact in three disjoint protection domains. Domain 1 has the power to operate on objects in domains 2 and 3, but domains 2 and 3 can neither access domain 1 nor each other. Domain 1 may be a collection of system monitors, while domains 2 and 3 may be for client communities that want to keep their data secure from other clients.

Figure 3-1. OBJECTS IN PROTECTION DOMAINS

Relationships between protection domains are defined by the virtual object machine as it is embodied in the **C++/P** compiler and in run-time support routines. Some domains may hold unilateral power over lesser domains; the more powerful domain can access the elements of the lesser domain, but not vice-versa (much like the kernel/user model); see Figure 3-1 for an example of domain relationships. Some domains may be completely disjoint, having no access to each others' insides (like processes in a multi-user time-shared system, or like domains 2 and 3 in the figure). An object may be in multiple domains at one time, and therefore can belong to several object communities.

A protection domain violation nominally causes a trap, and the trap handler is given information on the identity of the client and server involved in the transaction, their respective domains, and other useful data. The trap handler can take what ever action it deems appropriate to handle the infraction: abort the program, destroy the client object, print a message, ignore the infraction, *etc.*

One important use of protection domains is to build firewalls to protect the integrity and security of selected portions of the system. A system's domains may all be mutually disjoint, and can be thought of as being *enemies* of each other. This model can be compared to processes in a multi-user time-shared environment, except that all the power and flexibility of protection domains (dynamic domains, overlapping domains, *etc.*) are available. Any attempt to illegally cross a protection domain boundary causes a trap, and the trap routine can invoke recovery services or destroy the offending object.

3.3 System Services

It is often necessary to allow for a small number of exceptions to the protection model to allow access to an object's services (just like banks have teller windows for clients to transact business while keeping them away from the vault). In the kernel/user model, this is analogous to system traps that perform functions like I/O and system information retrieval. These services do rigorous argument checking to ensure that the client's request is reasonable and will not jeopardize the health of the system (for example, checking to see that the user of an I/O routine provides a buffer that is entirely inside memory owned by the user requesting the service). The identity of such services is known by the virtual object machine.

3.4 Use of Protection Domains as a Debugging Tool

Protection domains can be used to implement a debugging aid that can behave like a breakpoint or a data access trap. The virtual object machine is made to understand a domain called the DEBUG domain. Any object can be in the DEBUG domain in addition to being in its ‘‘normal’’ domain. The virtual object machine is set up so that a reference to anything in the DEBUG domain causes a trap. The trap routine can then pause execution or print out the values of object data; after it is done, execution can be transparently returned to the point of the trap.

3.5 Implementation of Protection Domains

Two questions had to be asked to implement protection domains:

1. Where is protection information kept?
2. Where is protection information enforced?

First, protection information could be kept with each object to define what operations it can perform on other objects. This approach was taken in operating systems like Hydra;[2] the mechanism was called *C-lists*. This approach makes it difficult to control an object’s accessibility at run time because the protection domain information for any given object is distributed. It can also become inefficient if the C-list is long (that is, if the current object is the client of many other objects) because the C-list is searched on every outboard object access.

A second approach would be to collocate protection information with the server. Each server would own information defining what domains were associated with that server.

Protection information could be kept in *capabilities*, programming entities that behave like object pointers except for the protection information they carry. Capabilities can be passed off from object to object; depending on the capability’s protection attributes, a user may or may not be able to make a copy of a capability. A capability acts like a single element of a Hydra C-list except that it is not in a list at all. It is like the approach where protection information is collocated with the object in that a capability stands in for a serving object.

All of these approaches were somewhat inappropriate for **C++/P** because of its view of domains, instead of objects, as the basis of protection. The information that had to be kept around was the mapping of objects onto domains, or vice-versa. Since there is no physical construct corresponding to a domain, domain information was kept inside of objects in a place inaccessible to the object itself. The compiler generated code that automatically arranged for the population of objects with domain information whenever they were declared or instantiated.

The other question is: who does the enforcing? The enforcing can either be done at the requesting end of a transaction or at the receiving end. Checking at the requesting end has the obvious fox-left-guarding-the-chickens problem, but checking at the receiving end also has its problems. If client `A` calls member function `B.C()`, a protection domain check can automatically be put into the code of `B.C()` by the compiler. But what if `A` references the variable `B.D`? The variable is not ‘‘active’’; that is, it has no associated object code that is executed when it is referenced. Putting the check at the receiving end is difficult without hardware support. **C++** therefore causes code to be generated at the requesting end to check for protection violations (see Appendix 1). The fox/chicken syndrome is avoided by the fact that it is the compiler that generates the checking code, and there is nothing the programmer can do about it.

The **C++/P** compiler gives system programmers a high degree of flexibility in defining and administering protection domains (see ‘‘Environment Administration,’’ below). Every system object could be protected as an object. Individual object data and member functions could be given their own protection domain attributes (Appendix 2).

The compiler currently has some hooks in it that allow users to shut down the generation of protection domain code in two steps. The **+NOPROT** option completely turns off generation of protection generation code. The **+NOMEMBPROT** option causes the compiler to generate code that only checks for protection domain violations at the object interface (membrane) level, foregoing checks for individual object elements. This can result in faster, more succinct code at the expense of less thorough protection checking.

3.6 The Need for Hardware Support

Our experiments with protection domains were just that—experiments to learn about protection models. Protection should ultimately be implemented in hardware instead of through code generated by the compiler. Without hardware protection, **C** pointers can be used by knowledgeable programmers to point at, read and modify the protection domain information in their own objects as well as in others. Ambitious programmers could write a compiler that didn't generate protection checking code at all. True protection could be realized when the protection domain analogy of tagged memory is put into hardware.

Hardware support can be applied to protection in other ways too. Protection domain violations could generate traps at the microinstruction level instead of at the architectural level of the machine.

Of course, having good hardware support will change the focus back to the compiler. A compiler that is knowledgeable about its object-based target can help with early problem detection as well as putting the power of object-based hardware at the programmer's fingertips.

4. ENVIRONMENT ADMINISTRATION

The **C++/P** compiler generates the code that enforces protection domains, so the compiler is the focus for much of the protection security work in our object-based systems.

First of all, every programmer is given a *license* to use the **C++** compiler. The license is an encrypted string of characters that is given to the user as a ''key'' to use the compiler. The decrypting algorithm is based partly on user-specific information, so licenses can not be ''stolen'' or given away by non-administrators. The user assigns the string value to an environment variable before invoking the compiler. The compiler decrypts the string and checks it for validity. The decrypted string points to a secure database that contains project and protection domain information for that particular license. The compiler reads information from the database and configures itself to understand protection enforcement and code generation for the appropriate project. The protection domain keywords themselves are kept in the database and are read in by the compiler to augment the scanner to understand them as tokens.

This arrangement has several nice properties. First, it allows users to work under any one of a number of licenses. Licenses can only be given out by an administrator who has access to a key forging program; the key forger recognizes only certain users and then only if they type in the right password. Every license can correspond to a different protection domain model. Once a license is given out, access to individual protection domains can be added and deleted on a per-license basis by modifying the database. Removing the file for a particular license takes away the owner's ability to develop any **C++/P** code.

5. CONCLUSION

The **C++/P** compiler gives the programmer all the power of object-based programming in **C++**, plus it adds the important object features of run-time protection and SmallTalk™-like object communication using messages. This combined object environment addresses many of the needs of systems that have stringent real time and recovery requirements, which have long field lifetimes subjected to frequent software updates, and which must be tolerant of software errors. These studies in object architectures point to the need to have hardware support for both protection and object communication.

Thansk and acknowledgments to Francis Leung, Jim Leth, Jim Vandendorpe, Tom Burrows, and Bjarne Stroustrup, for their contributions to the ideas in this paper.

IHP-55437-JOC-joc J. O. Coplien

REFERENCES

1. Dahl, O-J. et al. *SIMULA Common Base Language.* Norwegian Computing Center S-22, Oslo, Norway, 1970.
2. Wulf, W. A., R. Levin and S. P. Harbison. *HYDRA/C.mmp: An Experimental Computer System*, McGraw-Hill Company, 1981.

CODE GENERATION FOR TASK OBJECT COMMUNICATIONS

The **C++/P** compiler generates code to automatically construct and send a message when a user invokes the member function of a task object. A task object is an object whose object type (class) name is `Task`, or which has an ancestor named `Task` in the class inheritance hierarchy. Other (non-task) objects continue to interact using standard **C++** protocols that behave like **C** function calls.

Consider the code fragment:

```
#include "om.h"

class server : Task {
    server * next;
public:
    char _Buf[20];  /* big enough to hold message text */
    query(short);
    spool(char *, int);
    server(int);    /* constructor */
    ~server();      /* destructor */
};

. . .

server * device = new server(1);
device->spool("hello world", 11);
```

C++ reduces this to the C equivalent:

```
struct server {
    struct server * server_next;
    char _Buf[20];
};

extern server_query( );
extern server_spool( );
extern server__ctor( );
extern server__dtor( );

. . .

struct server * device;
device = server__ctor( NULL, 1 );
server_spool( device, "hello world", 11 );
```

The call to *spool* is translated into a real C function call whose argument list has been augmented to include a pointer to the object that is to perform the operation.

To translate the *spool* call into a message, the compiler first generates code to package the function's arguments into an ASCII string. This string and a corresponding decoding string are then passed to a common message handling interface for dispatching. The generated code is:

```
struct server {
    struct server * server_next;
    char _Buf[20];
};

extern server_query( );
extern server_spool( );
extern server__ctor( );
extern server__dtor( );

. . .

struct server * device;
device = (
    sprintf( auto_this->_Buf , "%ld", 1 ),
    _OM_send(&om, 0, 0, "server", "_ctor", auto_this->_Buf, "%l")
) ;
(
    (
        sprintf( auto_this->_Buf, "%s", "hello world" ),
        sprintf( auto_this->_Buf, "%s:%ld", auto_this->_Buf, 11 )
    ),
    _OM_send(&om, 0, device, "server", "spool", auto_this->_Buf, "%11s%l")
);
```

The first call to `_OM_send` causes memory to be allocated for the object; calls to ‘‘_ctor’’ (constructor) functions do not block or send messages, but return values immediately. The second call will cause a message to be queued to the object pointed to by *device*.

The two calls to *sprintf* cause the literal arguments ’`"hello world"`’ and ’11’ to be formatted as colon-separated ASCII strings in the user-supplied buffer `_Buf`. The user must supply a buffer that is big enough to hold any message that can be generated by calls from the object where the buffer is declared. This buffer is then passed to `_OM_send` along with other arguments. The first argument to `_OM_send` (‘‘`&om`’’) is for the **C++** convention of always passing an object’s address as the first argument to its member functions. The next argument (0 here) will usually contain protection domain information for the invoking object. The third argument (‘‘device’’) is an identifier for the receiving object; it is the quantity returned by the constructor and behaves much like an object capability. The fourth and fifth arguments argument (’`"server"`’ and ’`"spool"`’) are used to identify the receiving object type and member function name. The sixth argument contains the encoded arguments, and the last argument is a format string that describes how to decode the argument string. It behaves much like a *printf* format, using formats having the following significance:

%c	character
%*NN*s	literal string of length *NN*
%d	integer
%l	long
%h	short
%*NN*a...	aggregate (structure or class) of *NN* elements, which follow
%p...	pointer to type ...
%*NN*v...	vector of *NN* elements of type ...

The routine _OM_send performs low-level protocol tasks (like sending the message over an Ethernet™ network or depositing the message in an internally maintained list for later dispatching). The ‘‘virtual object machine’’ (in some projects, the Object Manager) unwinds the encoded argument string onto the stack and into heap storage, as appropriate, and then enters the requested routine.

CODE GENERATION FOR PROTECTION DOMAINS

The **C++/P** compiler generates code to check for protection domain violations at run time and to call a trap routine if an illegal access is detected. This code generation affects the layout of aggregates in memory, code for instantiating objects, and code that uses elements of an object. For sake of example, consider the following **C++/P** code fragment:

```
class X {
public:
        int i @ PERM;
        int j @ REST;
        void f(int, int);
} @ PROT;

newelement = new class X;
```

The compiler adds some "invisible" fields to the class that are used to hold protection domain information. These fields are bytes in the current implementation for most of our prototypes. The resulting layout appears as follows:

_prot_domain	X	X	X
_prot_X_i	X	X	X
_X_i			
_prot_X_j	X	X	X
_X_j			
_prot_X_f	X	X	X

The field `_prot_domain` is used to store information about the protection domain of the object "membrane"; it is the first domain checked when anyone tries to access the object. There are also byte protection fields for each of the elements of the class, whether they are data elements or member functions. This implementation mimics object machine tagged memory architectures to a certain degree.

This framework sets aside space for protection domain information, but the object is populated with protection domain data at instantiation time. The compiler translates the `newelement` assignment statement into the following:

```
newelement =
(
    ((class X *)_temp) = (class X *)malloc(24),
    ((class X *)_temp)->_prot_domain = PROT,
    ((class X *)_temp)->_prot_X_i = PERM,
    ((class X *)_temp)->_prot_X_j = REST,
    ((class X *)_temp)->_prot_X_f = ((class X *)_temp)->_prot_domain,
    (class X *)_temp
);
```

The *malloc* call allocates storage for the object off of the system heap. The variable `_temp` is assigned a pointer to this area. The subsequent assignment expressions are evaluated in turn to populate the object with protection domain information.

The protection domain code is actually a little more involved than this because of interactions with the initialization code in object constructors.

The compiler also generates code to compare the protection domain of a referencing object to that of a referenced object, and generates a trap if an infraction is about to occur. Consider the expression:

```
this.element = target.element
```

C++/P translates this into:

```
(
    * (
      (
          (current.domain <= this.domain)
            ? NULL
            : _trap(this.domain, current.domain, ...)
      ),
      ( & this.element )
    )
)
=
(
    * (
      (
          (current.domain <= target.domain)
            ? NULL
            : _trap(target.domain, current.domain, ...)
      ),
      ( & target.element )
    )
)
```

(For the sake of simplicity, this example only shows protection checks at the object interface level, and not those generated to check protection at the element level). The code generated for an expression is invariant with respect to its position relative to an assignment operator; the clever use of the ‘‘`*`’’ and ‘‘`&`’’ operators makes this possible.

Protection domain code generation for task objects differs substantially from that of the previous example. For `Tasks`, an object’s protection domain information is passed along with the message that contains a request for an operation to be performed on another object. The ‘‘virtual object machine’’ checks the protection information and can take appropriate action if an illegal access is detected.

One current implementation of the `_trap` routine makes use of UNIX® Operating System signals to give the user control over domain traps. The `_trap` routine saves the ‘‘state’’ of the trap—that is, the file and line number of the infraction, the domain of the initiator and of the operand, *etc.*, in global variables. It then checks to see if the user has redirected a predefined signal to execute their own trap routine. If the signal has been previously defined, then `_trap` sends that signal to itself, causing the user routine to be entered. The user can perform ‘‘fault recovery’’ or, finding that the trap is just a DEBUG trap, can dump useful information or interact with the user.

APPENDIX 2
PROTECTION DOMAIN SYNTAX

Protection information is specified in **C++** programs using *domain specifications*. A domain specification may be attached to a name, or to a class declaration, or to invocations of `new`. Used in the appropriate context, domain specifications associate protection information with the corresponding construct. For example, assume we started with the following declaration:

```
class X {
      int xi1, xi2;
      char * xstr;
};

class X xc;
```

To associate a protection domain with the class instance *xc*, we would change the last line:

```
class X xc @ PROTECTED;
```

which would put *xc* in the domain PROTECTED. If, in addition, we wanted to put additional protection on, say, *xi1* and *xstr* inside of *xc*, then this construct could be used:

```
class X xc @ PROTECTED (xi1=RESTRICTED, xstr=PERMANENT);
```

This puts some of `xc`'s elements in more sacred domains. If this is not done, they take on the same protection domain as the class itself (in general, the same as the enclosing scope).

It is also possible to associate protection domains with classes themselves. (Class *instances* like `xc`, are *objects*: they take up memory and contain information. Classes, on the other hand, are types, and are much like structure templates.) If a class declaration has domain specifications, then all objects declared to be of that class's type will take on the specified domain specifications. For example, we can make all objects of type class `X` to be in the PROTECTED domain:

```
class X {
      int xi1, xi2;
      char * xstr;
} @ PROT;
```

(PROT is an abbreviation for PROTECTED, as is REST for RESTRICTED and PERM for PERMANENT). It is also possible to put the class elements in individual domains, either by using the ex post facto construct:

```
class X {
      int xi1, xi2;
      char * xstr;
} @ PROT (xi1=REST, xstr=PERM);
```

or by associating the domains directly with the elements:

```
class X {
      int xi1 @ REST, xi2;
      char * xstr @ PERM;
} @ PROT;
```

These latter two are equivalent.

Protection domains on class types can be overridden when an object is instantiated by using the constructs described above. For example,

```
class X xc @ (xi1=PERM, xi2=REST);
```

would put `xi1` in the PERM domain, `xi2` in the REST domain, leave `xstr` in the PERM domain, and the variable `xc` itself to be the same domain as the enclosing scope.

Object pointers do not have protection domain information directly associated with them. Instead, the domain specification is attached to the "new" that generates the corresponding object, and the protection information stays with the object. For example:

```
class X * xp;
xp = new class X @ PROT (xi1=PERM);
```

Automatic object pointer variables inside functions can also be instantiated with `new` and an optional domain spec:

```
main()
{
    class X * xp = new class X @ PROT (xi1=PERM);
    class X * xp2;
    int i;

    xp2 = new class X;

        . . .
}
```

One ‘‘gotcha’’: vector type class elements have protection domain information associated with them as follows:

```
class X {
      int xi1, xi2;
      char * xstr;
      char carray @ REST [5];   /* five-character array */
} @ PROT;
```

It’s easier to remember if you think of the protection domain being associated with the name.

CONTENTS

LIST OF FIGURES