EXPERIENCES BUILDING AN OBJECT-ORIENTED SYSTEM IN C++*

Peter W. Madany and Roy H. Campbell and Panos Kougiouris
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue, Urbana, IL 61801

Abstract

This paper describes the methods and classes that we defined and the tools that we built to augment the C++ language so that it would better support the construction of an object-oriented operating system. To motivate the development of these facilities, we briefly describe *Choices*, an object-oriented operating system that runs on bare hardware, and its requirements for object-oriented language support. Together these facilities provide the automatic deletion of unreachable objects, first-class classes, dynamically loadable code for classes, and class-oriented debugging. Because of our experience building *Choices*, we advocate these features as useful, simplifying and unifying many aspects of object-oriented systems programming.

1 Introduction

Choices[Campbell 87] is an object-oriented operating system written in an object-oriented language (C++[Stroustrup 86]). It supports an object-oriented application interface based on objects, inheritance, and polymorphism. User and system objects can be created and deleted dynamically. In this paper we describe our experiences programming the system and the tools we built to take advantage of C++ in the construction of a large object-oriented system. These tools include:

- the automatic deletion of unreachable objects from various environments,
- run-time exposure of classes and the class hierarchy,
- facilities to load dynamically new subclasses and code into the kernel or a user program,
- debugging facilities for objects and classes that are independent of the compiler vendor, machine, and environment.

Our research efforts are directed towards developing an object-oriented operating system and not towards language design. We seek to use the object-oriented facilities of a language like C++ to achieve our goals of building an object-oriented operating system. From our perspective, an object-oriented systems programming language should include a minimal set of efficient encapsulation, inheritance, and polymorphism mechanisms that can be used to build the other concepts required in developing a system. Although C++ itself is not a minimal language,

*This work was supported in part by NSF grant CISE-1-5-30035, NASA grant NAG 1-163, and IBM agreement MHVV2431.

we have found it efficient[Russo 90]. Despite the lack of certain object-oriented facilities in the language, we have been able to construct these facilities from more primitive mechanisms. We believe our experiences may be useful to future object-oriented language designers and builders of other object-oriented systems.

Section 2 introduces the *Choices* operating system and provides the motivation for developing automatic deletion of unreachable objects, first-class classes, run-time loadable code, and portable debugging facilities. The next four sections describe our implementation of each of these features. Section 7 summarizes our experiences.

2 The *Choices* Object-Oriented Operating System

Choices has, as its kernel, a dynamic collection of objects. System resources, mechanisms, and policies are represented as instances of classes that belong to a class hierarchy. For programming convenience, the root of the hierarchy is class *Object* and the classes of the hierarchy are represented at run-time as *Class* objects.

The system has over 300 classes and 78,000 lines of source code. *Choices* runs on bare hardware: the Encore Multimax, the Apple Macintosh IIx, the AT&T WGS-386, and the IBM PS/2. It supports both uniprocessor and multiprocessor architectures. *Choices* is not a reimplementation of UNIX nor does it require the services of a UNIX kernel.

All entities in the operating system are modeled as objects and include: system processes, user processes, regions of memory, files, and hardware devices like CPU's and disk controllers. The application/kernel interface is defined by method invocations from objects in user space to objects in kernel space. In user space, a kernel object is represented by an *ObjectProxy*[Russo 91]. An *ObjectProxy* is created dynamically if the protection policy of the kernel object is not violated.

A *Choices Process* executes in a *Domain*. The *Domain* maps a collection of logical data stores or *MemoryObjects* into a virtual memory[Russo 89]. The *MemoryObject* provides random access to the contents of the data store. A *MemoryObject* can be memory-mapped into virtual memory. Direct access to the contents of the *MemoryObject* is provided by using the virtual memory hardware mapping mechanism. Each memory-mapped *MemoryObject* can be paged to its own backing store using independent paging algorithms.

Persistent data is implemented as a backing store that is loaded as a *MemoryObject* into virtual memory when needed. Persistent objects are encapsulated persistent data items defined by a class hierarchy. They are activated on demand and deactivated automatically[Campbell 90]. Persistent objects may store references to other persistent objects.

The mechanism underlying persistent data is derived from an object-oriented file system model. The model separates access methods from implementation details. *MemoryObject* subclasses support access to data stored on disks, files, or primary memory. The data may be formatted in various ways including the representations used to support UNIX 4.3 BSD, MS-DOS, and UNIX System V file systems[Madany 89].

Security is provided by a combination of access rights maintained by the file system, individual objects, and a user/system protection mechanism. Kernel objects are protected by supervisor state and by the virtual memory hardware. Method calls to kernel objects from user applications cannot be performed unless the user has been granted permission to access the object. That permission is provided as an *ObjectProxy*. Once provided, all method calls to the *ObjectProxy* are trapped and converted to method calls to the kernel object. During the trap, they are checked for validity by a kernel protection mechanism. Non-kernel objects are mapped into the virtual memory space of the application. Their data segments may be shared between applica-

tions using shared virtual memory. Once access is established, method calls can be made directly from the application to the object.

3 Automatic Deletion of Unreachable Objects

In *Choices* all resources, hardware devices, data, and processes are represented within an object-oriented model of the system. The system requires efficient mechanisms to create and delete objects shared by many *Domains*. For example, several *Processes*, some of which are in different *Domains*, may open the same file for read/write access. The file system represents the persistent data in the file as a *MemoryObject*. A *FileStream* instance provides a read/write interface to an application program and implements the UNIX notion of a sequential byte stream and current position within the file. Each application process opening a file uses a *FileStream* as its interface. However, it may share the interface with other processes in the same *Domain*. In both cases, *MemoryObjects* and *FileStreams* cannot be deleted until all the processes that access them have closed the file. Automatic deletion of *MemoryObjects* and *FileStreams* when they are no longer required simplifies code and eliminates system programming errors.

Although ad hoc reference counting and other garbage collection techniques can be employed to implement the features of the operating system, these mechanisms arise so often in an operating system implementation that we have chosen to embed them within the C++ base classes that we use to support our software development. Several object-oriented languages, including Smalltalk[Goldberg 83], allow objects to be constructed explicitly and delete them implicitly. In C++, however, objects must be both constructed and deleted explicitly. This places a significant burden on the programmer, since in all but the most trivial object-oriented systems, it is impossible to determine at compile-time when each object should be deleted. Two methods of determining at run-time when to delete objects are garbage collection and reference-counting[Goldberg 83].

3.1 Garbage Collection Algorithms

Garbage collection algorithms start with a set of objects, called *root* objects, that are by definition in use. Next, they determine all objects that are reachable by any chain of references from the root objects, and then they delete all objects that were not reachable. Reference counting algorithms keep a count of the number of pointers to each object. When an object's reference-count reaches zero, the object is deleted.

Garbage collection is often considered expensive, though some algorithms have negligible overhead per object deleted[Ungar 84]. Even efficient algorithms, however, may have unacceptable interactive performance, though some algorithms are designed specifically to avoid this problem[Baker 78]. Reference counting overhead is proportional to how often pointers to objects are stored.

Garbage collection algorithms for languages like Smalltalk can effectively determine all objects to delete; reference counting algorithms for most languages, however, usually fail to detect unreachable objects that belong to "pointer cycles." To implement an effective garbage collection algorithm, one must be able to determine the location of objects and which words of storage are pointers to objects[Seliger 90].

We chose to use reference counting for two reasons: first, reference counting has predictable space and time overheads, and second, run-time information about the location of C++ objects and pointers is not readily available.

3.2 Reference-Counting Functions

Having chosen to use reference-counting for automatic object deletion, we designed a set of classes, methods, and programming conventions to automate the process as much as possible.

All objects in the system that require automatic deletion inherit reference-counting behavior from class Object, which has an integer reference-count and five member functions related to this behavior: Object, reference, unreference, noRemainingReferences, and ~Object.

Object() The constructor for class *Object* initializes the object's reference-count and inserts the object in the global object table. The global object table enables storage leak detection (by displaying all objects that still exist at system shutdown) and the display of members of classes at run-time (see Section 4).

reference() The public member function reference atomically increments the object's reference-count; it must be called each time a pointer to an object is stored. To increase performance, reference is designed so that it can be made an inline function.

unreference() The public member function unreference atomically decrements the object's reference-count and calls noRemainingReferences if the object's reference-count reaches zero. It must be called each time a pointer to an object is overwritten. To increase performance, unreference is designed so that it can be made an inline function.

noRemainingReferences() The protected, virtual member function noRemainingReferences(), which should only be called by unreference, calls the object's destructor by deleting "this." This function can be overloaded to define statements to be executed *before* the object's destructor is called; no statements, however, can be executed *after* the destructor has been called.

~Object() The protected, virtual destructor for class *Object*, which should only be called by noRemainingReferences, removes the object from the global object table. To avoid premature deletion of objects, all subclasses of class *Object* must define protected destructors.

These five methods provide an effective and flexible mechanism that implements reference-counting behavior of objects, but they still place too heavy a burden on the programmer. This burden is the requirement that calls to reference and unreference functions be placed at *all* appropriate places throughout the code. Experience showed this requirement was too difficult to satisfy. To remove this burden, we chose to treat pointers to reference-counted objects as objects themselves.

3.3 Pointers as Objects

The *ObjectStar* class defines "first-class" pointers to Objects. Instances of *ObjectStar* have a traditional C++ pointer (`Object * _pointer`) as their only data member. Because *ObjectStar* defines no virtual member functions, its instances have no vtable pointer and therefore require the same amount of storage as a traditional C++ pointer. The *ObjectStar* class defines five types of member functions: constructors, destructors, dereferencing operators, assignment operators, and equivalence operators. We use *ObjectStars* wherever traditional C++ pointers would normally be used for member variables, local variables, global variables, and return values from functions.

Constructors and Destructors *ObjectStar* defines three constructors and one destructor:

```
ObjectStar();
ObjectStar( Object * );
ObjectStar( ObjectStar & );
~ObjectStar();
```

The first constructor initializes `_pointer` to zero; it allows programs to define arrays of ObjectStars, since the class of the elements of an array in C++ must provide a parameterless constructor[Stroustrup 86]. The second and third constructors store their parameter value in `_pointer` and call `_pointer->reference()`

if `_pointer` is non-zero. The destructor calls `_pointer->unreference()` if `_pointer` is non-zero.

Assignment Operators *ObjectStar* defines two assignment operators:

```
ObjectStar& operator=( Object * );
ObjectStar& operator=( ObjectStar & );
```

Both perform the operations of the constructor and then the operations of the destructor that takes the same type of parameter.

Dereferencing Operators *ObjectStar* defines two dereferencing operators:

```
operator Object *();
Object * operator->();
```

The `Object *` operator, which converts an *ObjectStar* to a traditional pointer, is intended to be used when passing pointers to objects as parameters to functions. Operator `->` supports the invocation of the member functions of the object pointed to by `_pointer`.

Equivalence Operators *ObjectStar* defines two equivalence operators:

```
friend int operator==( ObjectStar &, void * );
friend int operator!=( ObjectStar &, void * );
```

Operator `==` returns `true` if and only if the value of the `_pointer` member of the first parameter is equal to the value of the second parameter. Operator `!=` returns `true` if and only if the value of the `_pointer` member of the first parameter is not equal to the value of the second parameter.

3.4 Pointer Meta-hierarchy

To preserve the compile-time type checking features of C++, we developed a tool to build a meta-hierarchy of pointer classes that mirrors the hierarchy rooted by class *Object*. Each class in the meta-hierarchy defines constructors, assignment operators, and dereference operators analogous to those defined by *ObjectStar*. Neither a destructor nor equivalence operators need to be defined, since they can be inherited directly from class *ObjectStar*. Each subclass of *ObjectStar*, however, defines two additional constructors and two additional assignment operators:

```
SubclassStar( Object * );
SubclassStar( ObjectStar & );
SubclassStar& operator=( Object * );
SubclassStar& operator=( ObjectStar & );
```

Besides performing the operations of the similar functions described in Section 3.3, these constructors and assignment operators perform a downward pointer cast[1] and a run-time check (see Section 4) of the safety of the cast.

4 Classes as Instances

The motivation for representing classes as run-time instances in *Choices* arises from several requirements.

- The object-oriented application interface supports the dynamic creation and removal of objects in the kernel and user space. References to the objects may be passed from object to object. Class enquiry functions allow simplified programming, monitoring, and class-based, run-time controllable debugging.
- Instances of new subclasses of system abstract classes provide a mechanism to extend the application interface in a controlled but non-trivial manner. Classes provide query functions that return their superclass and subclasses, documenting the state and evolution of the system and providing a database of system facilities.
- Instances of a class can be listed providing a database of system services that conform to the protocol of the class.

[1] There are few cases in which a downward pointer cast is appropriate, one such case we have found is the retrieval of persistent objects from name servers and object stores.

- The classes of objects are represented at run-time as objects. This allows run-time checking[2] that the protocols used in messages to objects are consistent with their definition.
- The classes of persistent objects can be recorded as persistent objects.

4.1 Implementation

First-class classes or class objects are implemented in *Choices* using a class called *Class*. *Classes* are similar to the *Dossiers* described in [Interrante 90], except that *Classes* also support dynamic code linking and portable debugging. At run-time, every significant object in the *Choices* system is associated with a specific *Class*. The code given in Figure 1 defines the interface for *Classes*.

The `typedef` defines pointers to functions we call *addressable constructors* that initialize dynamically loaded class instances (see Section 5). The Class constructor takes a pointer to the *Class's* superclass, which is zero for the root of a hierarchy, and a character string name of the class. The constructor has an optional third parameter, which is a pointer to an addressable constructor. If a class does not have an addressable constructor or if its definition is deferred until run-time, then the third argument can be omitted. If an addressable constructor is supplied at run-time, a *CodeLoader* (see Section 5) may use the setConstructor operation to bind future invocations of the *Class's* constructor to invocations of a specific addressable constructor. The constructor operation provides a dynamically-bound constructor that will initialize an instance of the class corresponding to the *Class*.

The classOf operation returns a pointer to the *Class* of which the object is an instance. The isA

```
typedef ObjectStar (*CP)( Object * );

class Class : public Object {
                ...
public:
   Class( Class *, char * name, CP constructor = 0 );

   virtual setConstructor( CP constructor );
   virtual ObjectStar constructor( Object * );

   virtual ClassStar classOf();
   virtual int isA( Class * );

   virtual char * name();

   virtual ClassStar parent();
   virtual ClassStar child();
   virtual ClassStar sibling();

   virtual int displayMembers( OutputStream * );
   virtual int displayKindred( OutputStream * );

   virtual void setInitDebugMask( char mask );
   virtual void setInitDebugMessages( short msgs );
   virtual void setInitDebugObjects( int objects );

   virtual int debugMembers( char mask, short msgs );
   virtual int debugKindred( char mask, short msgs );
}
```

Figure 1: Class Interface Definition

[2] C++ provides safe and efficient compile-time type checking; this type checking can be, however, circumvented. Also, one can use run-time type checking to double-check compile-time type checking.

operation returns true if and only if the parameter is a superclass of the *Class*.

The class hierarchy is modeled as a tree of arbitrary degree but it is implemented as a binary tree. Each *Class* object has a parent (superclass), a child (subclass), and a sibling (the next class with the same superclass). By convention, a zero indicates the parent of the root of a hierarchy, the child of a *Class* with no subclasses, the sibling of a *Class* with no superclass, or the sibling of a *Class* at the end of its superclass's sibling list.

Using Smalltalk terminology, an object is a *member* of the class from which it was instantiated. An object is said to be a *kind* of the class of which it is a member, and of all the class's superclasses. The displayMembers operation displays on the *OutputStream* the names of all objects that are members of this *Class*. The displayKindred operation displays on the *OutputStream* the names of all objects that are kind of this *Class*.

The next five methods help support the portable debugging features described in Section 6. The setInitDebugMask, setInitDebugMessages, and setInitDebugObjects operations set the *Class's* initial Raid mask, message-count, and object-count to given values. The object-count determines how many future objects will be initialized with the current mask and message-count. If the object-count is zero, future objects belonging to this *Class* will be given zero masks and message-counts, regardless of their current values.

The debugMembers operation sets the Raid mask and Raid message-count of all existing members of the *Class*. The debugKindred operation sets the Raid mask and Raid message-count of all existing objects that are a kind of this *Class*.

4.2 Usage

An auxiliary program, **allClasses**, coordinates the definition and instantiation of *Classes*; it is run automatically by the *Choices* **make** files. To simplify the **allClasses** program, we adopt a programming convention that requires certain lines of code to be present in both the header (".h") and source (".cc") files of a class. For example, consider the class *Object*. It is defined in "Object.h" and implemented in "Object.cc". The lines added to define a first-class class for *Object* in file "Object.h" are:

```
extern ClassStar ObjectClass;

class Object ... {
                ...
public:
                ClassStar classOf();
                ...
};
```

The lines define a pointer to the *Class* that is assigned a value at run-time. The value is used by the classOf implementation shown below. The lines needed in file "Object.cc" are:

```
ClassStar
Object::classOf()
{
      return( ObjectClass );
}
```

4.3 Displaying the Class Hierarchy from the System Shell

Each *Class* is also bound to a name in the kernel's *NameServer*. A user of the system shell can request information about any *Class's* place in the hierarchy. The *Class* is looked up by name in the *NameServer*, and the requested information is displayed.

There are three commands for displaying class hierarchy information:

1. **ancestors**, which recursively displays the superclasses of the given class;

2. **descendents**, which recursively displays the subclasses of the given class; and

3. **hierarchy**, which recursively displays first the superclasses and then the subclasses of the given class.

An example of the **hierarchy** command, which shows both the ancestors and descendents of class *UNIXInode*, is shown below:

```
Choices> hierarchy UNIXInode
Object
   MemoryRange
     MemoryObject
       PersistentMemoryObject
         FileObject
           UNIXInode <<<
             BSDInode
             SVIDInode
```

4.4 Displaying Instances of Classes from the System Shell

All instances of subclasses of class *Object* are inserted into a global object table when constructed and removed from the table when deleted. *Classes* have access to this table and can iterate through the table searching for instances of the class that they represent.

The system shell uses the Smalltalk terminology both for displaying instances of a class and for Raid debugging. There are two commands for displaying instances of classes:

1. **members**, which displays all instances of the given class, and

2. **kindred**, which displays all instances of the given class and its descendants.

Examples of these commands follow:

```
Choices> members MemoryObjectPartition
MemoryObjectPartition[458100](:0:2)
MemoryObjectPartition[458040](:0:0)
2 instances.

Choices> kindred UNIXInode
BSDInode[466000](/)(:0:0:2)
BSDInode[466100](lost+found)(:0:0:3)
SVIDInode[45a180](/)(:0:2:2)
3 instances.
```

5 Adding Subclasses to a Running C++ Program

Choices provides an extensible application interface that permits the addition of new services to the kernel without rebooting the system. Implementing an extensible kernel as a dynamic collection of objects allows a minimal kernel to be defined for *Choices*. New features may be added to the kernel to customize it for applications or for the target environment. Existing software in the system may invoke methods on these new services if they are subclassed from an abstract class known to the system at compile time. New software added to the system may use all methods of the new services. C++ has neither dynamic linking of code nor dynamic binding for constructors. To simplify and generalize the mechanism involved, the base classes of the system were extended to support the dynamic loading and linking of new subclasses and their methods using a scheme that has been used by several other research groups[Dorward 90].

To access newly loaded code from an existing, running program, the running program needs a dynamic binding mechanism that maps messages (function calls) to invocations of the new code. C++ has an ideal mechanism for performing this dynamic binding: the virtual function table which is accessible through an object's *vtable* pointer. Once an object has been constructed, the member functions can be invoked through the object's vtable pointer. However, C++ constructors, which assign vtable pointers to objects at run-time, are *statically bound* and may not be called directly from an existing, running program. The problem to be solved in an implementation of dynamic, loadable C++ code is to allow existing programs to call the constructors defined in the newly loaded classes.

5.1 Dynamically-bound Constructors

Each class in the *Choices* system is represented by a *Class* object, described in Section 4, which

is accessible through *NameServers*. Our solution to invoking traditional static C++ constructors dynamically uses a constructor method, which indirectly invokes the traditional static C++ constructor and is defined for *Classes*.

We distinguish three interrelated constructor functions: traditional C++ constructors, which have names like Object::Object, addressable constructors, which we name in the manner Object-Constructor, and a dynamic constructor, which we name Class::constructor. Traditional constructors are no longer called directly by code that is designed to take advantage of the dynamic linking facility. However, they are still used to allocate and initialize heap storage.

An addressable constructor, which is implemented as a C++ function, is linked and loaded at the same time as the new class to which it will be bound. The linker can thus bind an invocation of the traditional constructor into the code for an addressable constructor. Unlike traditional constructors, addressable constructors return a zero in case of failure. A failure may occur if the class of the parameter to the addressable constructor was inappropriate (because a run-time check failed.)

A dynamic constructor is implemented by the Class::constructor function. It provides dynamic linking to the traditional static constructor function by an invocation of the addressable constructor to which the static constructor has been linked. When the code for the new class is to be loaded, a *Class* object is created to represent the new class. When the code is loaded, the linker installs the address of the addressable constructor in the *Class* object's _constructor instance variable. When the dynamic constructor of the *Class* object is invoked, it, in turn, invokes the addressable constructor stored in the _constructor instance variable. The addressable constructor then invokes the static constructor.

In detail, the following steps are performed by the dynamic constructor:

1. If the _constructor variable is zero, it creates an instance of class *CodeLoader* (described below) and initializes it with the name of the *Class*.
2. Since the *CodeLoader* object performs all its work when created, it then deletes the *CodeLoader* object.
3. If the _constructor variable is still zero, then it fails by returning zero.
4. If the _constructor variable is non-zero, it calls the addressable constructor, which calls the traditional constructor. The return value of the traditional constructor is also returned by the addressable constructor, which is then returned by the dynamic constructor.

5.2 The CodeLoader

Choices defines a subclass of the abstract class *CodeLoader* for each type of object file format used by the project. The first such class that we implemented supported the Common Object File Format[Gircys 88]; thus the subclass is named *COFFLoader*. The *CodeLoader* constructor performs the following operations:

1. open a file that contains the symbol table for the running program.
2. look up the name of the class with the undefined constructor in a *NameServers* that maps class names to file names.
3. if the file name is found, open the corresponding file, which should be a relocatable, unstripped module that includes the code for the class and for closely related classes.
4. allocate memory and read code from file.
5. resolve undefined symbols in the loaded code with the symbols found in the running program's symbol table.
6. relocate symbols in the loaded code, basing them on the memory address at which the code is loaded.

7. create *Class* objects for any loaded classes that do not already have them.

8. install addresses of addressable constructors defined in the loaded code into corresponding *Classes*.

5.3 Evaluation

By making loadable C++ classes available in the support for *Choices*, parts of the operating system can be greatly simplified. For example, the *Choices* file system supports UNIX 4.3 BSD, System V, and MS-DOS files. File system components that are needed can be loaded when a request is made to access a certain type of file. Thus, neither the kernel nor user space is penalized for the flexibility that is provided by the file system. A further benefit of this approach in the file system is that it makes loading the class methods for persistent objects that are activated simple to implement. Further, user-defined persistent objects may be added at any time to the collection of persistent objects in the system using the same mechanism. The C++ language could have provided improved support for loadable objects if invocations of constructors could be linked at run-time.

6 Portable Debugging

Debugging the kernel of an operating system is not a pleasant task, even if it is object-oriented. The problem is compounded by a lack of a kernel debugger that can report errors in terms of the objects and classes of which the system is composed. Debugging systems for C++ do exist, but most of them are proprietary, making them unsuitable for use in our activities. Further, our system is implemented on a wide variety of architectures. Because of compiler availability, for some systems we use the Gnu **g++** compiler, and in other cases we use AT&T's **cfront**. The Gnu symbolic debugger **gdb** allows symbolic debugging of programs, but this system only gives rudimentary support to the debugging of classes and instances. The lack of debugging facilities and non-uniformity of the debugging environment encouraged us to build debugging into the system as a feature that we call Raid, which is the brand name of some household pesticides.

Raid implements portable, object-oriented debugging in *Choices*. It is implemented as a set of methods, instances, and statements that are included in the *Choices* source code and operate by printing the values of expressions at run-time on the *Choices* console. Because the debugging facilities exist in objects at run-time, it provides a programmable debugging interface. Various debugging features can be turned on or off dynamically at run-time using *Choices* applications, like the *Choices* command interpreter or standard debugging tools like **gdb**. Further, a library of debugging utilities has been built up to provide more sophisticated debugging tools. Raid provides a much more flexible debugging environment than one based on placing diagnostic print statements in the code because it is integrated with the *Choices* object-oriented architecture.

6.1 Raid Masks

Raid supports five different categories of debugging statements for either existing instances or future instances of specific classes or classes and their descendents. These debugging categories are shown in Figure 2 along with the six bit masks that are used to encode them in Raid. The categories reflect common debugging problems that have arisen in *Choices*.

Masks can be combined (using a logical sum) in *interactive debugging commands.* For example, the *Choices* system shell (command interpreter) includes the **odb** or **cdb** statements (see below) that, with the mask `"RaidConstructor | RaidReference"`, can be used to view constructor, destructor, and reference counting method invocations. For convenience, masks can either be specified as bit masks or as names (option-

Category	Raid Mask Name	Bit	Mask
constructor/destructor functions	`RaidConstructor`	0	1
reference/unreference/noRemainingReferences functions	`RaidReference`	1	2
public member functions	`RaidMethod`	2	4
protected and private member functions	`RaidProtected`	3	8
detailed information within any type of member function	`RaidDetail`	4	16
all of the above	`RaidAll`	0–4	31

Figure 2: Raid Debugging Masks

ally abbreviated) enclosed in double quotation marks. Code that is instrumented with Raid statements use a single named mask to indicate under what conditions the information from the instrumentation is to be displayed. The names used are given in Figure 2.

6.2 Raid Implementation

C++ offers both the well-known C-style printf statement that some C++ developers prefer or the new C++-style << operator. Raid supports both means of printing debugging information:

- Raid(mask)() supports printf-style debugging, and
- Raids(mask) supports <<-style debugging.

Raid has a simple implementation — a pair of macro definitions, which use the logic given in Figure 3.

6.3 Instrumentation

Raid does not automate the inclusion of debugging statements; the developer must still insert these statements at useful places within the code. From experience we have adopted the convention of inserting Raid statements at three places within member functions: on function entry (to display the arguments with which the function was called), before each return statement (to display the return value(s) of the function), and between or within blocks of code (to display detailed information about the current function invocation).

Because Raid is part of the *Choices* system, one must exercise some restraint to avoid writing expressions in debugging statements that might have *Choices* side-effects. For example, the "++" or "--" operators will cause side-effects if used within Raid statements.

Constructor Instrumentation The beginning of each constructor should call the initDebug method. This initializes the object's mask and message count with the values set in its *Class*. Constructors may also include Raid statements on entry and exit, like the examples given below for member functions.

Function Entry Instrumentation Function entry can be instrumented by inserting a Raid statement at the beginning of the function. This Raid statement should be labeled with the appropriate mask. For example, one can instrument the beginning of the following member function:

```
ReadFileStream::read( char * buffer, int length )
```

using the Raid mask "`RaidMethod`" with either of the following Raid statements:

```
Raid( RaidMethod )
    ( "%N->ReadFileStream::read(%x, %d)\n",
      this, buffer, length );
Raids( RaidMethod )
     << this << "->ReadFileStream::read("
     << asHex( buffer ) << ", " << length << ")\n";
```

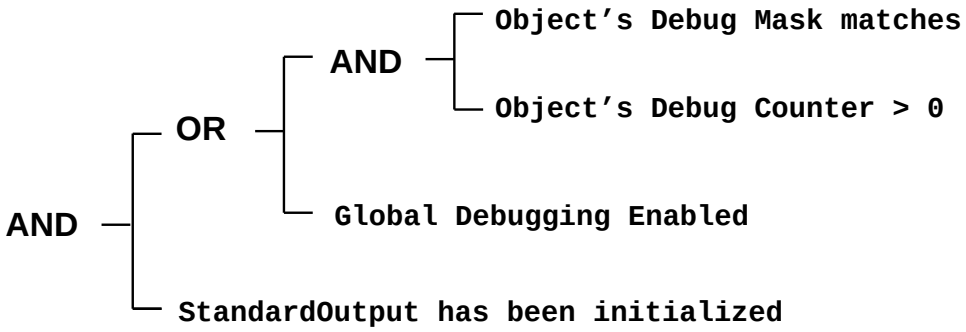

Figure 3: Raid Logic

The various Raid masks allow functions to be classified into the categories given in Figure 2.

Function Exit Instrumentation Function exit can be instrumented by inserting a Raid statement before the return of the function. For example, before the return statement:

```
return( bytesRead );
```

the code can be instrumented by a Raid statement with either of the following forms:

```
Raid( RaidMethod )
    ( "%N->ReadFileStream::read returns %d\n",
      this, bytesRead );

Raids( RaidMethod )
     << this << "->ReadFileStream::read returns "
     << bytesRead << "\n";
```

The mask for entry and exit should match.

Detailed Debugging Complex code requires a more detailed debugging instrumentation that provides information on the status of variables within a function. A developer may add other Raid statements throughout a function body. The "`RaidDetail`" mask is used for this purpose.

6.4 Controlling Raid output from the System Shell

Raid output can be controlled from both the *Choices* shell and from kernel or application code. This section describes the control of Raid from the system shell. There are two commands for controlling Raid debugging:

- **odb**, which turns on debugging for existing objects belonging to a given class, and
- **cdb**, which turns on debugging for future objects belonging to a given class.

The commands allow debugging information to be selected on a class basis for particular categories of instrumentation:

```
Choices> odb {member|kind} class mask [msgs]
Choices> cdb {member|kind} class mask [msgs [objs]]
```

For both commands, a mask of zero will turn off debugging. For example, the following command turns on debugging for the public methods of all the currently instantiated members of *ReadFileStream*:

```
Choices> odb member ReadFileStream "RaidMethod"
```

and the next command turns on all the debugging of all the currently instantiated members of *FileStream* and its subclasses. It will cause the display of the next 10 messages for each object:

```
Choices> odb kind FileStream "RaidAll" 10
```

Experiences with debugging *Choices* led us to add a Raid feature for examining particular activities. The **cdb** command allows instrumentation to be turned on for objects that are created as a result of some newly started activity. For example, the following command turns on constructor and reference count debugging for all future objects instantiated from class *BSDInode*:

```
Choices> cdb member BSDInode "RaidCon|RaidRef"
```

The count parameters may be used to turn on all debugging for 10,000 messages for each of the next, say, 3 objects instantiated. For example, the following command turns on debugging from instances of class *FileStream* and its subclasses:

```
Choices> cdb kind FileStream "RaidAll" 10000 3
```

6.5 Controlling Raid from Kernel or Application code

Debugging instrumentation can be controlled from C++ code by three operations on *Objects*:

1. setDebugMask(char mask)
2. setDebugMessages(short messages)
3. setName(char * str)

The setDebugMask operation sets the object's Raid mask to the given value. A mask of zero will turn off debugging for the receiving object. The setDebugMessages operation sets the object's message-count to the given value. A message-count of zero will turn off debugging for the receiving object. An object must have a non-zero mask and message-count before it will print debugging messages. The setName operation sets the object's "nickname" to a copy of the given character string. The nickname is printed out with each Raid statement.

In *Choices* objects may be debugged using Raid in either system or user space. This is facilitated by the *ObjectProxy* mechanism that makes invocations of method calls to kernel and user objects transparent to the calling program.

6.6 Using Raid with a Symbolic Debugger

Although Raid is useful by itself, it becomes a more powerful and flexible tool when used with a symbolic debugger. Advanced symbolic debuggers allow debugger commands to be executed at breakpoints and permit messages to be sent from the user to the objects of the program at run-time. The implementation of the symbolic debugger we use for system programming is independent of the *Choices* system console and so it can be used to debug the early stages of a *Choices* port to a new architecture. More importantly, the symbolic debugger allows the user to interact with Raid at run-time, based on the execution of specific breakpoints. Debugging information output can be selected by setting masks and counters interactively. This facility allows the behavior of a class hierarchy or instance to be examined at a specific point in the program. In boot code, it also allows Raid to be used interactively without input from the console.

For instance, using a symbolic debugger, one can define a macro that enables debugging in a region of a source file delimited by a starting and an ending line. The macro sets a breakpoint at the beginning of the region and another one at the end of the region. It also associates debugger commands with each breakpoint that turn on or off the global Raid debugging switch and continue execution. The whole process is invisible to the user who sees only the output of the Raid statements inside the region. This approach allows control of debugging on a method or even individual Raid statement basis.

7 Summary

During the construction of the *Choices* operating system, we discovered that several facilities can be of considerable importance and can lead to great code simplification in developing an object-oriented operating system. The facilities are not supported in C++, the *Choices* implementation language. However, by careful organization, we could introduce the facilities by building a suitable set of base classes and tools. The facilities we found of value are:

- the automatic deletion of unreachable objects,
- the representation of a class by an object at run-time,

- the dynamic loading of new subclasses and code, and
- support for debugging classes and instances at run-time.

The effect of these facilities was to speed up debugging and simplify the code of many parts of the operating system; for example, the file system, application interface, name-servers, and persistent object code.

In conclusion, we would like to draw the attention of language designers to the provision of these facilities in an object-oriented systems programming language. It remains an open question whether these facilities are best provided directly by the language or the language should support the construction of these facilities. Our experience with C++ shows that some of the mechanisms needed to implement the facilities could not be programmed directly in the language. Instead, we had to build tools to augment the C++ language.

References

[Baker 78] Henry G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, 1978.

[Campbell 87] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, Santa Fe, New Mexico, November 1987.

[Campbell 90] Roy H. Campbell and Peter W. Madany. Considerations of Persistence and Security in Choices, an Object-Oriented Operating System. In *International Workshop on Computer Architectures to Support Security and Persistence of Information*, pages 12.1–12.14, University of Bremen, Federal Republic of Germany, May 1990. The proceedings also to appear in the Springer-Verlag Lecture Notes Series.

[Dorward 90] Sean M. Dorward, Ravi Sethi, and Jonathan E. Shopiro. Adding New Code to a Running C++ Program. In *Proceedings of the USENIX C++ Conference*, pages 279–292, San Francisco, California, April 1990.

[Gircys 88] Gintaras R. Gircys. *Understanding and Using COFF.* O'Reilly and Associates, Inc., Newton, Massachusetts, 1988.

[Goldberg 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[Interrante 90] John A. Interrante and Mark A. Linton. Run-time Access to Type Information in C++. In *Proceedings of the USENIX C++ Conference*, pages 233–240, San Francisco, California, April 1990.

[Madany 89] Peter W. Madany, Roy H. Campbell, Vincent F. Russo, and Douglas E. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 311–328, Nottingham, UK, July 1989. Cambridge University Press.

[Russo 89] Vincent Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '89*, pages 267–278, New Orleans, Louisiana, September 1989.

[Russo 90] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: A Case Study. In *Proceedings of the USENIX C++ Conference*, pages 103–114, San Francisco, California, April 1990.

[Russo 91] Vincent F. Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, January 1991.

[Seliger 90] Robert Seliger. Extended C++. In *Proceedings of the USENIX C++ Conference*, pages 241–264, San Francisco, California, April 1990.

[Stroustrup 86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1986.

[Ungar 84] David Ungar. Generation Scavenging: a Non-disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1984.