Supporting Truly Object-Oriented Debugging of C++ Programs

James O. Coplien
AT&T Bell Laboratories
cope@research.att.com

Abstract

Most debuggers do not support an object-oriented debugging model. A debugger should be able to provide the view that each object is an independent entity with its own breakpoint behavior. We also would like the debugger to plant a breakpoint on the ‘‘right’’ member function when a polymorphic identifier is involved. The technology used in most C++ implementations does not support these features as well as the rich run-time environments commonly provided for symbolic languages. This paper introduces the need for such constructs, and presents algorithms that can be used to implement them in the framework of common symbolic debuggers.

1: Introduction

Though debuggers were among the first serious C++ programs written,[1] they are among the most anemic tools in most C++ environments today. Some vendors now offer good debuggers for specific hardware and operating system platforms, but applications building on niche platforms are often left at the mercy of their indigenous C debugger. Many developers use a hybrid C/C++ platform for C++ development, and are force to use a debugger with a C heritage to debug C++ code. In the pioneering days of C++, when most of the user base was tied to **cfront**-based compilers, installations that were serious about debugging taught their C debuggers some of the rudiments of C++. While the growing C++ market will increase the likelihood for more environments to enjoy full-fledged debuggers that support C++ language features, there will always be projects wishing to build C++ debugger support into their C debuggers for niche platforms.

Most C debuggers (or for that matter, debuggers for most procedural languages) support many of the same debugging constructs: planting breakpoints, examining variables, and dealing with scopes and activation records. The C culture shares a common model of procedural debugging, but the C++ world does not yet have a comparably mature ‘‘model of C++ debugging.’’ This immaturity owes partly to the object-oriented nature of C++, but it also owes much to the implementation technology underlying most C++ implementations.

We can contrast the C model of debugging with the Smalltalk model. Smalltalk programs run in a rich environment full of information about the running code. A programmer can interrupt program execution at any time and look around at the state of the world. There is no debugger tool per se. Objects are self-describing and can be queried, exercised, and modified on-the-fly. Programmers interact with their programs at run time using the same constructs and abstractions they use during design and coding. In fact, the CRC card design technique is closely tied to the Smalltalk style of debugging by browsing an interrupted program, providing a ‘‘hypothetical programming’’ approach to design.[2] Exploring a program at run time is an important component of object-oriented design, and a good debugger can be an invaluable aid during such discovery.

C++, in the heritage of C, distances the symbol-rich compile-time environment from the run-time environment. For reasons of efficiency and compatibility with C, C++ objects are usually not self-describing. These differences in technology cause us to think of C++ debugging differently from the way we think of Smalltalk debugging. (In fact, the paper you are reading has been rejected at other conferences because the reviewers felt that C++ developers are doomed to debug at the assembly language level, so object-oriented debugging models for C++ are only of academic interest!)

Many features come to mind as we think about what it means to have a ‘‘model of C++ debugging.’’ We can list these features in an informal order of increasing sophistication:

- *Demangling*, which lets the programmer talk to the debugger using source identifier names;
- *Overloading*, so the developer can distinguish between multiple functions of the same name;
- *Scope*, to recognize object and member function scope; for example, class member function breakpoints;
- *The Actors model*, where we associate member functions with the objects on whose behalf they execute;
- *Coarse-grain debugging* which would allow you to simultaneously set a breakpoint on all member functions of a given class or object;
- *Inheritance*, so base class member functions can be treated as though they are part of the derived class;
- *Intelligent formatting of user-defined types* when displaying their contents;
- *Debugging inline functions*, by providing breakpoint capabilities for them;
- *Object-oriented programming*, so the programmer can interchangeably treat objects of derived types as though they were instances of their common base type.

We find many of these features in even the most rudimentary C++ debuggers, but support for these features becomes less widespread as we descend the list. Many debuggers have only addressed class scoping and disambiguation of overloaded function names. Few C++ debuggers fundamentally change the model of debugging from that of C, and the C++ programmer is left to manipulate their C++ program in terms of the intermediate C code generated by **cfront**, or in terms of a C model of the object code from the C++ compiler. A great debugger would let programmers walk around in their running program with the same facility as they manipulate their C++ source; an ideal debugger would let programmers think about their programs in terms of object-oriented *design* abstractions. Smalltalk’s environment lets the Smalltalk programmer approach that ideal; with work, we can make a C++ debugger smart enough to provide many of the same capabilities.

This paper discusses models for two aspects of object-oriented debugging: support for object autonomy, and support for genericity. The paper also details implementation strategies for each, with the hope that they will be useful to projects wishing to add C++ to their local C debugger. This paper draws from work on prototype and semi-production debuggers widely used for C++ development at AT&T.

2: Object Autonomy

To consider what is meant by debugging support for object autonomy, we can contrast run-time aspects of procedural and object-oriented languages. Programs in either kind of language have both a dynamic and a static structure. In procedural languages, the unit of program composition is the function. We seldom worry about a function’s existence until it is called and an activation record for it appears on the stack. These activation records capture the dynamic structure of a program, and their content is the focus of much of our debugging effort. In languages without recursion (such as FORTRAN), run-time mappings between procedures and activation records are straightforward, since any procedure has at most one activation record open. The situation is slightly more complex in C, because it allows recursion and a given function may have an arbitrary number of activation records. However, use of recursion is more often the exception than the rule, and though most good debuggers have constructs for selecting from among a given function’s activation records, they are seldom used.

The class is the major unit of program composition under the object paradigm. Member functions form the bulk of static C++ program structure, just as in C. Though member functions have activation records at run time, the run time focus is on objects rather than on local function data. Multiple objects of a given class might be extant, and it is important to the programmer to be able to distinguish between such objects and to be able to easily access any one of them. Compare this with how we view a procedural program, where multiple ‘‘instances’’ (activation records) of the primary programming abstraction (a procedure) are the usual case. This difference in views affects how a programmer debugs their program, and the debugger should support the prevailing view (Figure 1).

The difference between these views is one of emphasis and interpretation, not of implementation. At the level of design, the class view is better suited to modeling genericity; we will return to this in the next section. The object view, often called the Actors model, is better suited to modeling object autonomy.

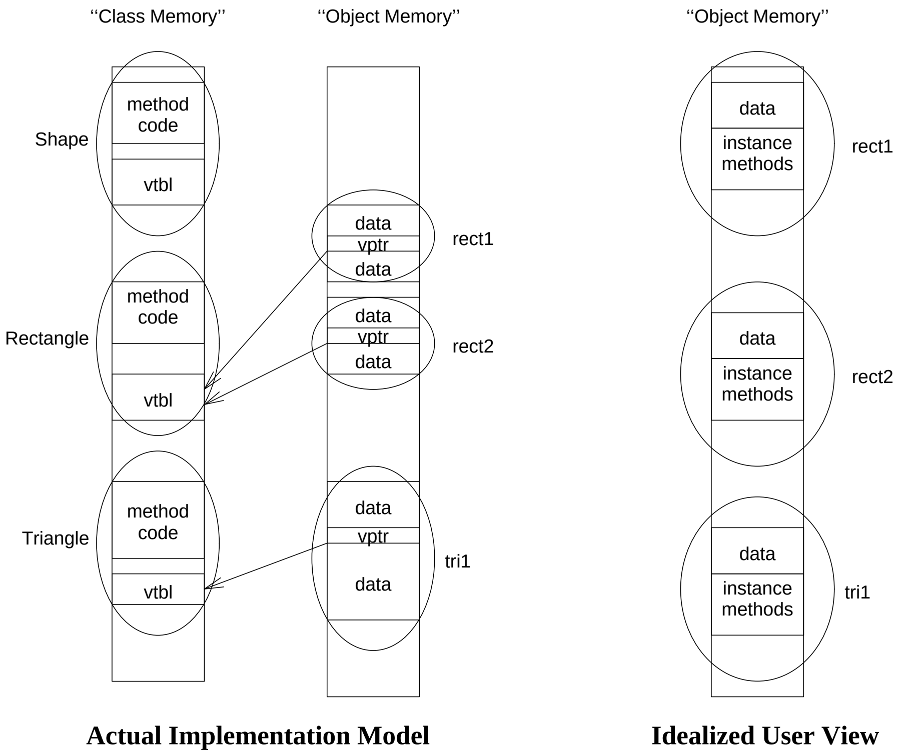

Figure 1. Object Implementation and User ''Actor'' View of Objects

A debugger command language might clearly distinguish between the class view and the object view. Consider the difference between:

```
when in Stack::pop { print "hello" }
```

and:

```
when in pancakes—>pop { print "hello" }
```

where `pancakes` points to an object of class `Stack`. The class form views the program in terms of its static structure, i.e., as a collection of classes, and the object form in terms of the objects making up its dynamic structure.

These two views are analogous to how one might view processes debugging in a multiuser programming environment. A simple operating system maintains a disjoint memory image for each invocation of a program, and each programmer has a complete copy of all text and data. If the environment supports shared text, special measures are necessary to allow one programmer to debug the code being executed by others. We would like to preserve both the illusion that each programmer owns their memory image, and the implementation advantages of shared text. The object-oriented analogue is for all objects to share the text of their classes' member functions, yet allow each object to be debugged as though it contained a complete copy of all its code.

We can easily teach a debugger to hide sharing details from the application programmer. The object breakpoint for `when in pancakes—>pop` can be formally defined as being equivalent to:

```
when in Stack::pop if (this == pancakes) { print("hello") }
```

where `this` is the anonymous object operand pointer argument passed to every member function. Such constructs were in fact used before the introduction of the object breakpoint construct in the debugger. The debugger's implementation of the object breakpoint construct can be obviously inferred from its class form.

The class view itself has two senses. The first sense is that of class methods (as opposed to instance methods), and is necessary in C++ to deal with member functions such as constructors. The other sense of

Class::Method understood by the debugger, particularly for instance methods, is that the associated operation (such as a breakpoint) applies to all instances of *Class* as though it had been applied to each individually.

Both the object and class views have been found to be useful in program debugging, and both approaches have been implemented in our prototype `sdb++` debugger using a syntax culturally compatible with `sdb`. One breakpoint of each of these forms may simultaneously be active on the same line, with the object version taking precedence over the class version when both apply.

We implemented this technique in the framework of the existing C debugger, `sdb`, on a UNIX† SVR3 Operating System base using the AT&T/USL C++ Compilation System. The algorithms can be outlined as follows:

1. The debugger parses the command, recognizing it as a breakpoint command.
2. The operand is analyzed, and is discovered to indicate the name of an object pointer, and a function related to that object. The operand is parsed into those two components.
3. The symbol table is searched for the variable containing the object pointer. From that symbol table entry, perhaps with some additional searching using algorithms common to most symbolic debuggers, the *structure tag* of that variable can be found. In many implementations of C++, this structure tag names the original class of the original C++ text.
4. The name mapping scheme is applied to map the class name and function name onto a single name (e.g., `_move_9Rectangle_`), which is the mechanism used by the C++ compiler to fold the nested C++ name space onto the flatter C name space.
5. The symbol table is searched for the generated function name, and the function address is extracted from the symbol table entry.
6. The *breakpoint header table* is searched for an entry containing this function address. This is a simple table containing elements whose fields are: a function address, a saved op code for the location where an associated breakpoint or trap instruction is planted, and a breakpoint list pointer.

   The breakpoint list pointer points to a list of one or more *breakpoint structures*. A breakpoint structure entry contains the address of an object which must be the operand of this function invocation in order for this breakpoint to fire; a pointer to the next breakpoint structure, and a flag indicating whether the breakpoint is an Actors breakpoint or a ‘‘regular’’ (non-Actors) breakpoint.

   If a header table item is found whose address field matches the function address, skip to step 10; otherwise, proceed to step 7.
7. Create a new header table entry, and store the newly generated function address in the appropriate field.
8. Save the machine instruction at the generated address in the appropriate place in the newly generated table entry.
9. The machine instruction at the generated address is overwritten with an instruction that will cause a breakpoint to occur.
10. Search the breakpoint structure list for a matching entry; if one is found, this is a redundant breakpoint and gets exceptional handling (error or warning).
11. Create a new breakpoint structure. Designate this as an Actors breakpoint.
12. Taking the name of the variable containing the object pointer, go to the symbol table and find its address.

---

† UNIX is a Registered Trademark of UNIX† Systems Laboratories, Inc., a subsidiary of Novell, Inc., in the U.S. and other countries.

13. Go into the target process at this address, and retrieve the contents there. The result is the address of the object (operand) of interest.

14. Store that address in the appropriate field of the newly created breakpoint structure.

Now, when a breakpoint fires, the debugger searches the breakpoint header table for an entry with a matching address field. The debugger can also reach into the target process and retrieve the value of the address of the current operand—it is the value of the variable this from the current activation record. In the list associated with the identified breakpoint header table entry, the debugger searches for a match with that address. If such a match is found, the breakpoint is processed by keeping control in the debugger; i.e., not returning control to the application process except to temporarily restore the original op code and step the program over it, and then restore the trap instruction. If such a match is not found, then the original instruction is replaced and stepped, the breakpoint trap is replaced, and execution of the application process is resumed.

A limitation of this model is that breakpoints cannot be associated with variables before program execution, but can be associated only with extant objects after execution has begun. A breakpoint is not associated with an *identifier*, but with some *object*, though an identifier is used to indicate the object to which the breakpoint applies.

3: Support for Genericity

Genericity, and in particular, run-time type support, is a key concept in object-oriented programming. C++ supports this form of genericity through inheritance and virtual functions. This genericity makes it possible for a client of a group of objects to address any of them through a single identifier declared in terms of their base class. The client code sees only the base class interface and is thus insulated from changes to the form of derived class objects and even from addition of new derived classes.

We would like our debugger to give us the same derived class transparency. After all, the person using the debugger is likely the one who wrote the application code, and it is unreasonable to expect the programmer to know, at run time, the exact class of an object created from a hierarchy. To provide this transparency, the debugger must allow the programmer to communicate breakpoint and inspection requests in terms of a base class identifier, and resolve them itself in terms of the actual type of the object as determined at run time. That is, we want the debugger to be able to handle virtual functions with the same power and flexibility as understood by the C++ compilation system and the code it generates.

3.1: Type fields in disguise

How can the debugger determine an object's type? The naive answer would be to look in the symbol table. Unfortunately, most interesting objects in a C++ program are allocated from the heap, so they have no address that can be translated to an identifier, and hence to a type, using symbol table information. Another possibility would be for the debugger to retrieve the object's type from its memory image. Unfortunately, C++ objects have no type field information, at least none that is easy to find. In this respect C++ is a minimalist language, maintaining data structures as close to C as possible. The presence of a gratuitous type field would upset assumptions about the size and layout of class data. A compiler might annotate virtual function tables with type name strings, but most compilers do not: such strings would consume memory space, and would be useful only for classes having virtual functions. It is the lack of an explicit type field that makes this an interesting problem, and is why this is an issue in C++ and not in, say, Smalltalk.

In fact, the compiler does deposit a type field of sorts into each object whose class contains virtual functions—it must do so for the virtual function dispatching mechanism to work. Associated with each class is a table, commonly called the *vtbl*, most of whose content is function dispatching data. The compiler lays down an instance of this table for each class, and arranges for class constructors to deposit a pointer to this table in every object of that class. [3] This field might be viewed as a type field whose offset in the object can be determined from the class of the object.

In the polymorphic case, a derived class object is accessed through an identifier declared in terms of the base class (a reference or a pointer to the base class). The debugger is asked to manipulate the object (for example, plant a breakpoint) with respect to the identifier declared in terms of the base class. The base class’s *vtbl* pointer offset can be used even in the context of derived classes: The compiler guarantees that the same offset is preserved along the entire derivation chain.

3.2: Finding the function address

Now that the *vtbl* pointer offset is in hand, how do we find the address of the function? There is potential ambiguity if derived classes override the base class function having the same name as the one of interest. At this point, the debugger knows the name of the function that is to have a breakpoint associated with it, and it knows the address of the *vtbl* for the member function’s class. If the debugger knew the class of the object of interest, and the function name, it could determine the breakpoint address directly from the symbol table. However, it does not know the class name at this point. Identifying the class’s *vtbl* is not sufficient to identify the class, because a derived class may share its base class’s *vtbl* if it overrides none of its base class virtual functions.

There is a roundabout way of determining the breakpoint address without ever having to know the name of the class, taking advantage of another property of *vtbl*s. Consider all classes in a given inheritance tree. Then for any function *foo*, the *vtbl* index at which *foo*’s entry appears is constant. All classes in an inheritance hierarchy maintain a constant mapping between function names (or, more precisely, function signatures) and their offset into the *vtbl*. We need to make one other assumption, which is that the requested function appears in the signature of the class corresponding to the identifier known to the debugger as the object’s handle. This is a safe assumption: If it did not, the user would have no business addressing that function in terms of the given variable, and would be admonished with an error message.

We now gather information to support the algorithm that follows. The symbol table can be scanned for functions whose addresses match the addresses in the base class *vtbl* entries; i.e., those associated with the identifier supplied to the debugger by the user. When a match is found between the address of symbol table entry and a *vtbl* entry address, we can tabulate the *vtbl* entry’s index with the symbol table entry’s name. Most symbol table formats support the gathering of such information without the overhead of a linear search.

3.3: The complete algorithm for virtual function breakpoints

Now, we have our table of name/*vtbl* index mappings, the name of the desired function, and the locations of the desired object and, by consequence, of its *vtbl*. We can look up the *vtbl* index for our desired function from the table. Finally, we can use that index as an offset into the object’s *vtbl*, and find the address of the function in that entry. At that address, we may plant a breakpoint.

Here are the details of the algorithm for the UNIX `sdb`-based debugger, given that context; they are also depicted in Figure 2.

1. The debugger parses the command, recognizing it as a breakpoint command.

2. The operand is analyzed, and is discovered to indicate the name of an object pointer, and a function related to that object. The operand is parsed into those two components.

3. The symbol table is searched for the variable containing the object pointer. From that symbol table entry, the structure tag (class name) is derived. This symbol table entry, generated by the compiler, contains compile-time known properties of the symbol, such as its type and its address in memory.‡ While the base type of the pointer is compile-time knowable, object-oriented languages allow such pointers to validly point to an object of any subclass of its declared base type, so something declared to point to a `Shape` may validly point to a `Rectangle` or a `Circle` at run time. So the class identified by this step is that corresponding to the pointer, *not* to the actual object it points to.

‡ For automatic (stack) variables, the symbol table usually contains an address field that corresponds to the symbol’s offset within the activation record of the function in which it is declared; for external symbols, it is an address in the program’s logical address space.

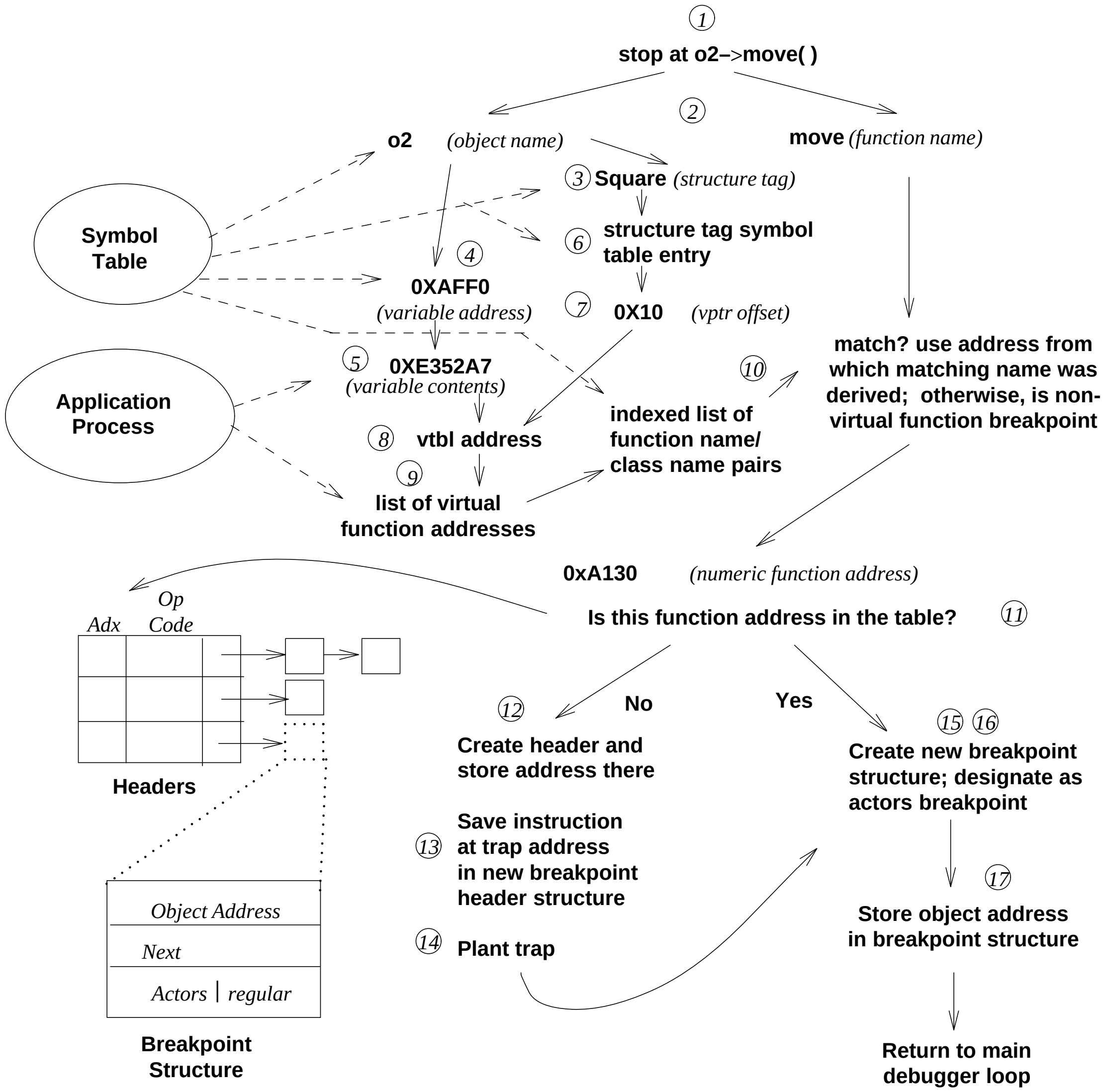

Figure 2. Virtual Actor Breakpoint Processing

4. From the same entry, extract the address of the operand variable.
5. Go into the target process at this address, and retrieve the contents there. The result is the address of the object (operand) of interest.
6. The symbol table is searched for an entry describing the fields of the structure named by the tag (class name) found above.
7. From that entry, the offset of the virtual function table pointer (`vptr`) is extracted.
8. Add this offset to the address of the object (operand) retrieved above, yielding the address of the word pointing to the `vtbl` associated with this object.

9. Retrieve the word at this address in the application program image; it is the address of the virtual function table (`vtbl`) for this object's class. The virtual function table is basically a list of pointers to functions; the index into such a table for a function of any given name is the same for all such tables and all such functions in classes participating in a derivation hierarchy whose root contains a function of that name. In reality, virtual function table entries contain additional data supporting multiple inheritance, which are ignored for the moment here.

10. The basic approach is to go through the virtual function table one element at a time, extracting function addresses from successive elements. Since function addresses are unique within a program, and function names are unique, there is a full, unambiguous and unqualified mapping back and forth between addresses and function names. However, in an object-oriented language like C++ that uses C technology for intermediate steps of the compilation process, the function names generated at the intermediate level may consist of two parts encoded together into a single name, those two parts being the class name and the function name.†† The encoding is reversible; the class and function names can be reconstituted unambiguously from the encoded name. We want to find the virtual function table entry whose function name component matches the function name specified by the user in step 2. This algorithm uses the object pointer supplied by the user, and information available from the object at run time, to deduce which function in the class hierarchy should be selected. In detail, the algorithm iterates over the virtual function table, and for each element does the following:

    a. Extract the function address for this entry.

    b. Do reverse symbolic resolution on the address; i.e., turn the address into a function name. This can be done by a linear scan of the symbol table (i.e., the task is tractable), but most debuggers build internal data structures to support doing this in a more efficient way; the details aren't important.

    c. Compare the function name component of the name/class pair thus generated with the name generated in the name parsing process from step 5 above. If there is a match, yield the address of the function and exit the loop.

    d. If the end of iteration is reached without finding a match, then this is *not* a virtual function Actors breakpoint, but a non-virtual function Actors breakpoint, and the ordinary Actors algorithm described earlier should be applied. Normally, the algorithm described here is applied first and, on this failure, the Actors algorithm is entered at the appropriate point (e.g., step 4) to avoid recalculation of data.

11. The address of the virtual function is now in hand. The breakpoint header table is searched for an entry containing this function address. If a header table item is found whose address field matches the function address, skip to step 15; otherwise, proceed to step 12.

12. Create a new header table entry, and store the newly generated function address in the appropriate field.

13. Save the machine instruction at the generated address in the appropriate place in the newly generated table entry.

14. The machine instruction at the generated address is overwritten with an instruction that will cause a breakpoint to occur.

15. Search the breakpoint structure list for a matching entry; if one is found, this is a redundant breakpoint and gets exceptional handling (error or warning).

---

†† The full signature (argument types, as well as the function name) is usually thus encoded into the internal (''mangled'') name by C++ translation systems.

16. Create a new breakpoint structure. Designate this as an Actors breakpoint.

17. Store the object address in the appropriate field of the newly created breakpoint structure.

4: Conclusion

The breakpoint techniques described here have been introduced in a number of prototype debuggers inside AT&T for a half dozen development platforms, and have been widely distributed throughout the company. The amount of effort needed to convert an existing debugger to use these techniques varies, but is a few staff-weeks on the average.

We have only anecdotal insights on how these techniques are used on AT&T projects. The Actors style breakpoints seem toenjoy frequent use, with generic breakpoints seeing somewhat less use. Part of this is due to the heavy use of data abstraction, but lesser use of inheritance, in projects actively using the debuggers. Debugger features supporting virtual functions and inheritance will likely see more use as understanding of the object paradigm deepens and spreads in the development community.

This algorithm was constructed in close collaboration with Tom Williams at Bell Laboratories. Discussions with Harold Bamford and Tim Born were also useful to refine our understanding of object-oriented debugging needs.

References

1. Cargill, T. A. ‘‘PI: A Case Study in Object-Oriented Programming.’’ *SIGPLAN Notices 21(11)*, November 1986, pp. 350-60.

2. Personal communication with Kent Beck and Ward Cunningham, 1993.

3. Ellis, Margaret A., and B. Stroustrup. *The Annotated C++ Reference Manual*. Reading, Mass.: Addison-Wesley, ©1990, sect. 10.5.