Portable Run-Time Type Description for Conventional Compilers

Sheetal V. Kakkad, Mark S. Johnstone, and Paul R. Wilson
Department of Computer Sciences
The University of Texas at Austin
and
Somerset Design Center
Motorola, Inc., Austin TX
Sheetal_Kakkad@email.mot.com, johnstone@acm.org, wilson@cs.utexas.edu

Abstract

Many useful programming language extensions and system support libraries require knowledge of the locations of fields within objects at run time. Examples include orthogonal persistent object stores, precise garbage collectors, data structure picklers, and parameter marshaling schemes.

For clean and efficient implementation as libraries, these systems require run-time knowledge of in-memory layouts of data objects, which is unavailable in most traditionally compiled and linked programming languages, such as C, C++, and Ada. Even the recently standardized *run-time type identification* (RTTI) feature in C++ is insufficient, because it describes only language-level features of the type hierarchy and not the compiler-dependent object layout decisions.

We present a facility for *run-time type description*, or RTTD, which extracts low-level layout information from debugging information generated by conventional compilers, and makes it available to user programs. We believe this to be the simplest and most portable approach to run-time type description, requiring no changes to existing compilers. In this paper, we describe the basic strategies and present details of our implementation for C++. We also sketch some extensions that we have implemented, including special treatment of C++'s virtual function table pointers to match persistent or foreign data objects with the actual code in a particular application.

Our implementation of run-time type description is freely available. It is in regular use with multiple operating systems and compilers, in both free and commercial products, including a high-performance persistent object storage system for C++ and a real-time garbage collector.

1 Introduction

Low-level information about the format of data objects is necessary for clean and efficient implementation of a variety of language extensions and run-time libraries. These include persistent object stores [ABC$^+$83, SKW92, AM95], data structure serializers (picklers), advanced foreign function call interfaces, parameter marshaling facilities for messaging and remote procedure calls, data structure browsers, data format conversion utilities, precise garbage collectors [Wil], advanced profiling, debugging, and tracing tools, and a variety of other systems support facilities. Traditionally, such features have relied heavily on programmer or compiler cooperation, or the use of special-purpose preprocessors or precompilers. However, we will show that each of these approaches has severe drawbacks.

In this paper, we describe a facility for *run-time type description* (RTTD), which provides implementation-level information about data representation by extracting it from the debugging information generated by conventional compilers. We believe that this is the most portable approach for supporting advanced run-time systems for a variety of applications. Note that while some languages provide type information at run time—such as C++'s *run-time type identification* (RTTI) facility—this information is insufficient for these applications, and should not be confused with RTTD. The information provided by RTTI is intended to support *language-level* operations such as safe subtype discrimination (narrowing casts or "downcasts"), while RTTD is intended to support *implementation-level* operations like locating pointers within objects for applications such as garbage collectors and persistent object stores.

Goals of our approach:

- *Efficiency*: All complex and time-critical processing is done at compile time, thus avoiding unnecessary overhead at run time.
- *Portability*: Our approach relies on standard compilers and formats for debugging information. No special operating system support is required.
- *Flexibility*: Our approach exploits existing mechanisms in a graceful way that is mostly independent of any particular combination of language, compiler, and operating system.
- *Ease of use*: Programmers need to make minimal changes to source programs even in uncooperative languages such as C++.
- *Elegance*: Our approach leverages off of standard practices in compilation and linkage process.

Structure of this paper. This paper describes RTTD at two levels of abstraction. First, we present a high-level description of our basic approach and contrast it with traditional approaches. Second, we present a detailed description of our implementation for C++. While we only show implementation details for C++, the basic approach and implementation are mostly independent of source language, and should be easily adaptable to different languages with a similar compilation and linkage model, such as C or Ada.

A note on portability. We use the term "portable" to denote the fact that our system works with standard compilers and operating systems. We do not mean that our system is portable in the "compile out-of-the-box and run" sense, but rather that it can easily be ported. It is therefore less portable than an ANSI C program, but much more easily portable than, for example, a "portable" compiler or operating system.

2 Problems with Traditional Approaches

There are two traditional approaches for extending existing languages:

1. modifying compilers to support the new facilities, and
2. constructing preprocessors and precompilers to translate the extended source language into the "base" language so that it can be compiled by standard compilers.

Unfortunately, both of these approaches have severe drawbacks. The construction of a high-quality compiler is a daunting task, and it is seldom worthwhile for the addition of a few language features. Porting such a compiler to all desired platforms, and maintaining it thereafter, is usually prohibitively expensive. On the other hand, preprocessors and precompilers often seem to be a simple solution to the problem, allowing portable source-to-source translation of an extended language into a base language. However, they also have hidden drawbacks making them impractical for many purposes:

- Preprocessors complicate error detection and reporting, and often evolve into full-blown *precompilers*, that is, essentially entire compiler front-ends which emit an intermediate representation that is the source language for another compiler.[1] Without sophisticated analysis and error detection, preprocessors are hard to use and programmers rightly avoid them. Precompilers perform more analysis, but reintroduce many of the problems of custom compilers, including difficulty of development and maintenance.
- Preprocessors and precompilers generally suffer from incompatibility problems because they do not "understand" the extensions provided by other preprocessors or precompilers. While it might seem simple to pipeline preprocessors that implement different language features, this approach does not work in general—constructs intended for one preprocessor may confuse preprocessors earlier in the pipeline. This is especially problematic for nested constructs, which may require repeated applications of each preprocessor; thus no fixed order or number of preprocessing steps is possible.
- Preprocessors that generate low-level layout information are *not* portable. In implementing system extensions, it is often necessary to take into account the decisions made by the underlying compiler, particularly with respect to the layouts of data objects. This is complicated by advanced type systems, because the source program may use parameterized types, abstract classes, and/or nested types. Furthermore, both the mapping from high-level types to concrete types[2] and the alignment and padding rules may vary from platform to platform. The preprocessor must infer the actions taken by a particular target compiler in order to determine the layouts of concrete types.
- Preprocessors are fragile because they are strongly tied to the source language. Introduction of new keywords and syntactic variations may confuse a preprocessor, even if those variations are unrelated to the preprocessor's actual job. Additionally, the preprocessor is likely to break in the face of vendor-specific extensions, or changes to a language that is undergoing gradual standardization (as was, until recently, the case with C++).

In general, because preprocessors reduce flexibility, we do not advocate their use, either for RTTD generation or for other situations. In contrast, using the compiler-generated debugging information bypasses most of these problems: our approach gets the relevant information from actual output generated by the compiler itself, rather than *inferring* it from an examination of the source code, greatly reducing the coupling between the compiler and the type descriptor generator.

3 Basic Design

Two problems must be solved to provide run-time type description:

1. constructing *type descriptor records* that describe layouts of types, and
2. associating appropriate type descriptor records with corresponding instances of these types at run time.

The first problem is easy to solve using standard debugging information. It is only necessary to compile source files with the compiler-specific debug flag turned on; the layout information can then be extracted from the debugging information generated by the compiler. If desired, the program can then be "stripped" of debugging information, or recompiled with the debug flag turned off, to avoid bloat and possible loss of optimization.[3]

Associating the type descriptor records with instances of types is somewhat more difficult. Conventional programming languages do not provide any explicit mechanism for associating compile-time information about types with run-time instances. For some languages, such as C, the simplicity of type construction and naming makes this task fairly easy. For others, such as C++ and Ada, a more sophisticated approach is necessary, involving some system dependencies.

Next, we describe our solution to these two problems, and how our approach fits into the standard compilation and linkage process.

3.1 Generating Type Descriptor Records from Debugging Information

Our basic approach is to extract type information from debugging information and construct a table that maps unique type names to the relevant layout information. This table is then made available to applications by linking in special access routines from a support library. Extracting the layout information from the debugging information is easy, because we require only a subset of the information required by most debuggers.

While the representation of debugging information may vary from platform to platform, there are a few standard formats which a tool can translate into a common type descriptor record format. However, different compilers may use slightly different conventions for representing some important information, and some tailoring may be required to adapt the system to other compilers. Alternatively, the system can be implemented using an existing code

[1] The original `cfront` translator from C++ to C is perhaps the best known precompiler.

[2] A *concrete* type is the actual type of an object instance, as opposed to an abstract or polymorphic (*parameterized* or *generic* or *template*) type, which may describe commonalities between multiple concrete types.

[3] In principle, the compiler could use different data representations when optimizations are enabled, but it appears that the representation of heap-allocated objects is invariant in all conventionally-linked systems—if this were not true, linkage problems would arise when linking code compiled with different levels of optimization.

base that can parse different formats of debugging information. For example, our type descriptor generator for C++ (described in Section 4) leverages code from gdb, the GNU debugger, which can parse debugging information for a variety of systems. Our type descriptor generator inherits gdb's generality, making many ports trivial. Other systems can be supported by implementing a relatively small module to extract debugging information in different formats,[4] or better yet by extending gdb to handle those formats.

A note about reliability is perhaps in order here. Many programmers are painfully familiar with errors in compiler-generated debugging information—the correctness of debugging information is often less critical than the correctness of the generated code, and sometimes compilers are released with broken support for debugging. However, our system relies on only a subset of debugging information—the type layout information—which is far more reliable than information that maps program counters to source lines, or maps local variables to registers, or other optimization-dependent information. We have found that the information we need is fairly reliably provided by the compilers we support so far, and we expect that to be the case in general. Occasionally, a version of a compiler may be broken in a relevant way, and we must advise using a different version.[5]

3.2 Associating Type Descriptor Records with Instances

Our strategy for associating type descriptor records with instances has two basic parts:

- *identify the concrete types at allocation sites*, and
- *record the type of each instance.*

The identification of concrete types may be tricky for some source languages, because the relevant information is often not directly available at the allocation site. For a language with a simple type system (for example, C), we can capture the type name at the allocation site using a macro front-end for the allocator. For more sophisticated languages, such as C++ or Ada, however, this does not work, because *macro expansion occurs too early in the overall compilation process*—before scoped, parameterized, and nested classes have been mapped to unique concrete types. Instead, we wait until *after* the code at the allocation site has been compiled to determine the concrete type of the objects to be allocated. Conceptually, this requires *backpatching* the allocation site—inserting a type identifier at each call site *after* it has been compiled—so that the allocator can record the object's type. Here, the *type identifier* is a unique key—which can be as simple as an integer index into a table—that refers to a type descriptor record.

As we will describe later, the identification of types and the corresponding backpatching is relatively easy. We introduce a level of indirection and leverage the compiler's type processing and the linker's natural name resolution. We introduce special variables, which we can refer to from allocation sites, that hold type identifiers for each type. The linker resolves references to these variables in the usual way.

3.3 Compilation and Linkage Model

To enable both the construction of type descriptor records and their association with instances, we carefully intervene in the compilation and linkage process, so that we can extract type information when it is available, create type descriptor records, generate code to create and initialize the special backpatching variables, and link together the object modules, initialization code, and support libraries.

The steps of the overall process (Figure 1) are as follows:

1. Compile source modules/files with the debugging flag enabled. In our C++ implementation, we provide a macro to be used at the allocation site to refer to the backpatching variables.
2. Extract type information from the debugging information in each object file. This is done by a stand-alone program, o2tdesc, provided with our system.
3. Merge type information from modules to be linked, to construct a table of type descriptor records for the application. This is done by another program, tdlink, which essentially "links" type information in a way analogous to linking object modules.
4. Construct a source file to define and initialize the backpatching variables used at each call site; this will initialize them with the appropriate type identifiers. This is done by a program called tnamemap, and the resulting file is compiled using the same compiler used for the application.
5. Build the application by linking all object modules, including the specially-generated object module (from the previous step) and a support library to load and access type descriptor records.

Notice that the construction of type descriptor records is on a per-module[6] basis, *following* compilation, and that linking of type information resembles linking of the application, but *precedes* the actual linkage of the application. The first ordering is necessary, because the type information is not available until after a module has been compiled. The second is optional, and is just an optimization—it allows duplicate elimination before linking. For dynamically-linked modules, it can be avoided by using separate tables of type descriptor records, at some space cost in redundant type information, or run-time cost in duplicate elimination.

Notice also that the type descriptor records can be generated as each module is compiled; standard makefiles can be used to automate the regeneration of type descriptor records for recompiled modules. At every compilation and linkage step, the cost is roughly proportional to—and generally much smaller than—the base cost of compilation and linkage.

4 RTTD for C++

In the above description, we glossed over some important language-specific issues regarding how unique type names are recognized in debugging information, and how the backpatching mechanism associates them with the allocation call sites. This process is intrinsically language and/or compiler-specific. In this section, we will describe how we implement the RTTD mechanism for C++.

Our approach exploits common practices in C++, but minimizes dependencies on any specific compiler implementation. The basic idea is to associate a unique variable with each allocation site, and ensure that this variable contains the appropriate type identifier value corresponding to the class/type instantiated at that allocation site. We optimize this approach by actually using one variable *for each unique type*,[7] and generating references to this variable at each site that allocates instances of that type.

[4] For example, we currently support IBM's C++ compiler for OS/2 using a 2000-line module rather than code from gdb. (In retrospect, this module is more general than necessary, and could probably be half as long.)

[5] For example, a few of the many releases of the GNU C++ compiler have broken our system, but this can be addressed by using a slightly older or newer version.

[6] For C++, a *module* is one source file, but the same scheme should work with other language implementations that use modules as units of compilation.

[7] In this paper, we use the term *type* generically to refer to both aggregate types (i.e., classes) and non-aggregate types (i.e., builtin types).

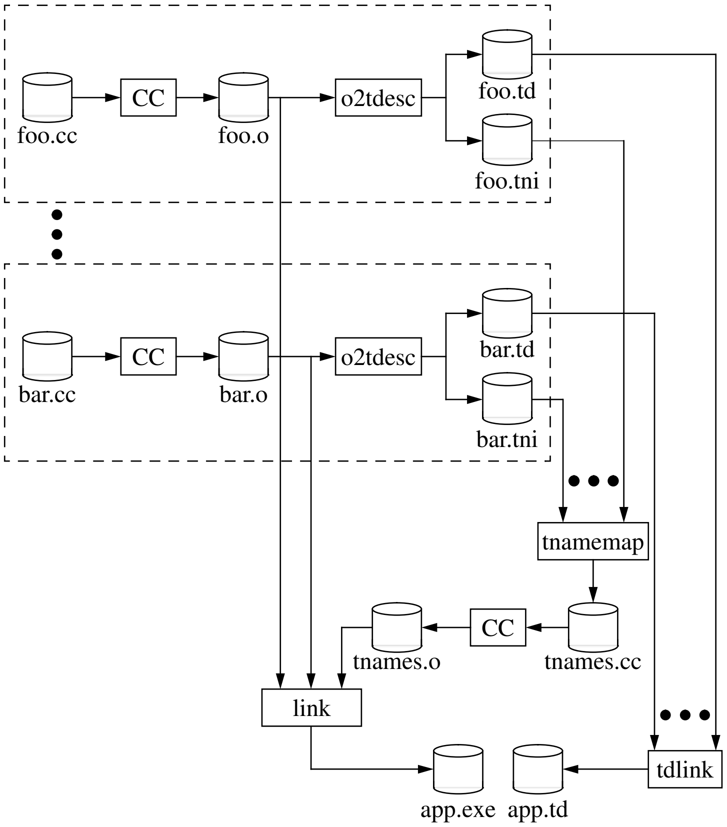

Figure 1: Compilation and linkage process

A naive approach for providing backpatching variables—which does not work in the general case—is to ensure that each type instantiated has a special class variable[8] which plays the role of the backpatching variable. However, this approach cannot handle builtin types (e.g., `int`, `float`, etc.) in C++ because the language does not allow the definition of class variables for non-aggregate types. Instead, we modify the solution slightly to achieve the same effect by using a *wrapper class* that is associated with each type that is instantiated. This wrapper class then contains the appropriate class variable (for backpatching the type identifier) and allows us to build a *one-to-one association* between the instantiated type and the type identifier. Furthermore, we implement the wrapper class as a parameterized (template) class, such that a concrete wrapper class can be automatically generated for every type that is instantiated. In general, a concrete wrapper class serves several roles:

- *It allows the easy and automatic creation of class variables for non-aggregate types.*

  C++ is a hybrid object-oriented language, and not all types are classes. Since only classes can have class variables, the use of a wrapper class provides a useful level of indirection in the type hierarchy. For each wrapped type, there is a unique wrapper class that holds the class variables constructed for that type.

- *It allows easy generation of references to the class variables used for backpatching.*

  The macro front-end to the allocator can simply instantiate the template wrapper class with the type, and construct a reference to the appropriate concrete wrapper class. The class variables associated with that class can then be accessed using the `::` (member reference) operator. Notice that instantiation of the parameterized wrapper class occurs at compile time, and the actual run-time reference to the class variable is fast.

- *It ensures that debugging information will appear for all types that are allocated using the allocator macro front-end.*

  C++ compilers may omit generation of debugging information for common predefined types; however, the type information for the wrapper class will include the type information for the wrapped type.

Notice that while a wrapper class is constructed for each type which needs type description, the resulting concrete wrapper class itself does *not* need to be instantiated to create any data objects at run time—it is simply a repository for class variables, and no instances are created. In other words, the parameterized wrapper class is a compile-time class generator, and the resulting concrete wrapper class has no instances.

4.1 Implementation Details

The previous section provided a high-level, overall description of the basic design. We now describe the major components of our implementation for C++.

4.1.1 The wrapper class

The parameterized wrapper class is of the following form:

[8] A *class variable* is a variable associated with a class, rather than instances of the class. In C++, class variables are implemented as static data members of the class.

```
template <class T> class TDWrapper
{
  private:
    T* wrappee;
  public:
    static int typeid;
    ...
};
```

The wrapper class declares a data member of type pointer-to-T, that is, whatever type it wraps. This ensures that the debugging information for the corresponding concrete wrapper class will include debugging information for the type it wraps. No instances of the concrete wrapper class are created, however, so this data member *never* exists at run time. The wrapper also declares, but does not define,[9] a static data member that will hold the type identifier for the wrapped type; this static data member is defined and initialized in a later step of the RTTD process.

4.1.2 Overloading the C++ new operator

The normal behavior of the new *operator* may be described as conceptually two distinct steps, even if the code generated by a particular compiler may not explicitly distinguish between them:

1. a compiler-provided operator new() *function* is called to obtain storage for the object, and
2. the constructor for the type is called to initialize the object appropriately.

C++ allows a programmer to overload the new *operator* for providing additional arguments to the operator new() *function*. We overload the C++ new operator to pass the type identifier information to a customized allocator via the operator new() function. It should be emphasized that C++ treats the new operator specially; when we overload it, we do *not* redefine all of the behavior of the operator. Instead, we are really overloading only the operator new() function (that is, the first step above), and the constructor is still called as the second step [Lip91].

We invoke the overloaded new operator using the so-called *placement syntax*[10] to pass the type identifier as an additional argument to the underlying allocator, to be stored with its bookkeeping information. Our customized allocator simply stores the object's type identifier in a hidden header word for each object, and supports mapping from pointers to an object (or to its interior) to the location of that header word.[11] Due to space limitations, we will not describe our allocator in detail here, but see [Kak97]. Most allocators can easily be modified to record the relevant information at little cost [WJNB95, Joh97].

4.1.3 The allocator macro front-end

We provide a macro front-end to the overloaded new operator to encapsulate the special syntax of invoking the overloaded operator. The macro definition expands to pass a reference to the type identifier (via the backpatching variable) as part of the placement arguments to the overloaded new operator, which then passes it on to the customized allocator. Any other arguments are passed to the constructor as normal constructor arguments.

The normal C++ syntax for calling the new operator is:
new *typename* (*args...*)
and the placement syntax is:
new (*p_args..*) *typename* (*args...*)
where *p_args...* refers to the placement arguments passed to the overloaded new operator. The syntax for calling our macro front-end is:
rttd_new(*typename*) (*args...*)
which resembles, but is not quite the same as, the non-placement syntax. Specifically, *extra* parentheses are required around the type name. This is the only syntactic change we require our users to make to their source code.

The extra parentheses in our syntax are necessary because the C/C++ macro preprocessor supports very weak semantics; it does not directly support passing multiple sets of arguments, or a variable number of arguments, or arguments not enclosed in parentheses. We fake having variable numbers of arguments by defining our macro such that it only recognizes the first argument (the type name) as an argument at all, while the remaining argument list (if any) is simply ignored and passed unmodified through the preprocessor. The compiler recognizes this (optional) argument list as the list of constructor arguments.

The rttd_new macro simply takes *one* argument, the type name, and uses it to construct a reference to the class variable of the concrete wrapper class corresponding to that type. A simplified version of the macro definition is of the following form:

```
#define rttd_new(type)  \
   new (TDWrapper<type >::typeid) \
      type
```

TDWrapper<type >::typeid refers to the class variable that holds the type identifier for the specified type. For example, the allocation call:

```
rttd_new(list<int>) (1,2,3)
```

will be translated by the preprocessor into

```
new (TDWrapper<list<int> >::typeid) \
   list<int> (1,2,3)
```

TDWrapper<list<int> >::typeid is passed as the placement argument to the overloaded new operator. The next expression, list<int> is the normal type specifier for the new operator. Finally, the last argument list provides the arguments to the constructor for type list<int>.

4.1.4 Generating backpatching information

So far, we have described the basic strategies for generating references to type identifiers at allocation sites—each allocation site refers to a class variable in the concrete wrapper class for the type being instantiated. Obviously, these class variables must be defined and correctly initialized to hold the appropriate type identifier values at run time, so that the entire scheme works as designed.

Our basic solution is to generate a C++ source file that contains the necessary definitions and initializations of these class variables; this file is then compiled and linked into the application, completing the association of type information with allocation sites. This step is easy, given our basic design, because we have already ensured that we can find all of the relevant debugging information in order to generate the type descriptor records. For each type descriptor record, we generate a line of code that defines and initializes the corresponding class variable. These lines are written to the source file which is then compiled and linked into the application.

A simplified version of this technique would be to initialize each class variable with the corresponding type identifier value—once we have generated the table of type descriptor records, the

---

[9] In C++, static data members are only *declared* in the class definition, and must be *defined* separately.

[10] This term was originally used to provide "placement" information for the object as an additional argument to the underlying allocator. It is now a misnomer because the same syntax can be used to pass *any* additional arguments to the allocator.

[11] For C and C++, this is necessary because pointers might not point to the start of an object. Pointer arithmetic may generate pointers to fields within a heap allocated object, and the usual implementation of multiple inheritance implicitly uses pointers to sub-objects.

type identifiers are available because they correspond to indexes in that table. Our actual implementation is slightly more complex, however, so that we can perform these lookups lazily and support advanced linkage techniques [Kak97].

4.2 Complications and Enhancements

The scheme we have described so far is oversimplified in several ways; our actual implementation is a little more complex. We now sketch some of the complications and enhancements. For a complete description, see [Kak97].

4.2.1 Forcing generation of debugging information

A smart compiler may avoid generating debugging information for classes that are never actually instantiated, or perhaps for classes that are instantiated but never operated upon by the application. Since we never actually instantiate concrete wrapper classes, it is possible that a compiler may optimize away the debugging information for these classes. In such situations, we can fool the compiler into thinking that the wrapper class *may* be instantiated at run time, though it never actually is. We do this by modifying the `rttd_new` macro definition to include an `if` statement whose condition expression is never true at run time. The `then` clause of this `if` statement—which is never actually executed—allocates an instance of the wrapper class and invokes an empty function on it. The `else` clause, of course, contains the "real" definition as described earlier. Because the `if` condition is not statically known to the compiler, it cannot optimize away the conditional expression. Hence, the compiler is forced to emit debugging information for the concrete wrapper class, which includes debugging information for the wrapped type.

4.2.2 Managing type names and virtual function tables for persistence and interoperability

For some applications, it may be desirable to avoid committing to particular type identifier values at compile time, and instead map type names to type descriptor records at *run time*. This capability is important for applications that deal with persistent data (such as persistent object stores and data structure picklers), or data communicated between applications (as in RPC systems), because `typeid`s in different applications generally will not match. Therefore, we use the *fully-qualified type name* string as a symbolic representation of type names. This representation is translated into actual type identifiers lazily and incrementally, using a slightly more complex allocation mechanism.

Similar issues arise in the treatment of C++ virtual function tables (VFTs). For persistence, the VFT pointers must be treated specially.[12] Most normal applications simply assume that VFT pointers have been initialized appropriately when an object is created, and do not change thereafter. However, the VFT pointers point to code *in the executable (text) segment of the address space*, not in the data segment like typical data pointers. Virtual function tables for the same classes may appear in different places in the executables of different applications using those classes; thus, simply saving and restoring virtual function table pointers does not work.

For data that may be operated on by multiple applications (or even by recompiled versions of the same application), we must save a symbolic representation of VFT addresses. This can be achieved by translating these addresses into "names" corresponding to the virtual function tables[13] when the data are saved or communicated. Later, these names can be translated back into addresses of the corresponding tables for the application that reloads or receives the data.[14]

Since this solution is similar to the problem of identifying type identifiers across different applications, we use the same basic approach as before. We translate VFT pointers into symbolic names by using link tables in the saving/sending application, and back into actual pointers using link tables in the restoring/receiving application. This solution is simple, but introduces a slight dependency on compilers' name mangling schemes—it must be possible to recognize VFT names as such, for example, by recognizing a leading substring like "`_vt$`."

4.2.3 Handling nested types

Our approach works well and is easy to use for most builtin types and normal classes. However, nested types/classes in C++ pose a special difficulty for the backpatching mechanism. When generating the initialization for the backpatching variables, we normally use forward declarations for the types (used as arguments to the parameterized wrapper class). This is necessary because the compiler will otherwise complain about unknown symbols—by providing a forward declaration, we are promising the compiler that, at link time, there will exist a valid symbol (in our case, a type name).

Unfortunately, C++ does not allow forward declarations for types and classes whose definitions are nested inside the definition of another class. We believe this is an unnecessary restriction imposed by the language. We work around this problem by requiring the programmer to provide a list of header files that contain the definitions of the nested types. This list is then included in the initialization source file, avoiding any spurious compilation errors.

5 Related Work

To the best of our knowledge, two other systems similar to ours have been developed, both independently; unfortunately, no published descriptions exist for either. Marc Shapiro and his collaborators at INRIA have developed a type description facility, also using some code from `gdb`.[15] It also appears that Object Design, Inc. (ODI) developed a similar facility for its ObjectStore persistent object storage system for C++, but no details of this proprietary system are available.

Our strategy for locating object headers from interior pointers is essentially identical to the one used in several garbage collectors, the best-known being the conservative collectors developed by Hans Boehm *et al.* [BW88].

The general strategy of making type description available to sophisticated run-time systems is present in most garbage-collected languages [Coh81, JL96, Wil]. Our contributions are to expose this information to arbitrary run-time libraries, which may be linked into applications in the usual way, and to support conventional off-the-shelf compilers for popular languages.

Several libraries have been designed to provide run-time type descriptor records for garbage collection (e.g., [AF94, Ede92b]). However, these libraries typically require programmers to provide methods that reveal pointer locations to the collector. Our system was designed to avoid burdening the programmer with such tedious and error-prone coding. Detlefs' library interface [Det92] is a little cleaner, but incurs extra overheads and restrictions because conceptually compile-time operations must be performed within the language at run time. Our system avoids this problem by performing

[12] A VFT pointer is a pointer to a table of function pointers, used for dynamic method dispatching. Each instance of a class with virtual functions has one or more VFT pointers, generated automatically by the compiler.

[13] Each virtual function table has an associated name—usually derived by combining the class name with some unique string used during name mangling—attached to it.

[14] Conceptually, this is a code linkage step—data objects contain pointers to code, which must be patched to point to the appropriate code when a different program is "linked" with the data. In programming language terms, we have implemented a simple dynamic linker for virtual function tables.

[15] Marc Shapiro, personal communication, May 1996.

most operations at compile time (or link time), at the expense of full portability—we must rely on debugging information and knowledge of the linking process.

Preprocessors for run-time type description have also been proposed (e.g., [Ede92a]). The best known is Interrante and Linton's *dossier* system [IL90], where a preprocessor-style dossier generator is used to create *Dossier* objects from the source code. Interrante and Linton also proposed that the language be extended to automatically generate a virtual function for each class to access the corresponding dossier object. For reasons described in Section 2, we prefer to avoid the need for a preprocessor or changes to the language.

6 Current Status and Future Work

We have implemented our run-time type description mechanism for C++. We currently use it in the Texas Persistent Store, a high performance persistent object store [SKW92] that uses pointer swizzling at page fault time [WK92] for address translation, and a real-time generational garbage collector for C++ [WJ93]. Our system works under several flavors of UNIX (SunOS, Solaris, Linux, Ultrix and others) with the GNU C++ compiler, under SunOS and Solaris with the GNU and Sun C++ compilers, and under OS/2 with IBM's CSet (VisualAge) C++ compiler. In principle, it will work with most operating systems (including Windows 95/NT and MacOS) and C++ compilers.[16] The implementations of the type descriptor generator (and the persistent store) are freely available at `ftp://ftp.cs.utexas.edu/pub/garbage/texas`.[17]

Scope and Limitations. Currently, we focus only on run-time type description for dynamically-allocated objects; this is sufficient for most applications. In the future, we intend to support RTTD for statically-allocated objects as well. Stack-allocated objects pose greater difficulties, and are also a topic for future work.

7 Conclusions

We have presented a facility for incorporating advanced run-time support systems into popular languages, using off-the-shelf compilers, linkers, and operating systems. Our approach is mostly portable, though it depends on debugging information formats and certain other aspects of compiler name management and linking. In many cases, ports to other platforms are trivial because compilers can emit debugging information in standard formats; in some cases, however, it may be necessary to write or modify code to recognize a new format.

We have found this library quite useful and easy to use in Texas, our persistent object storage system for C++, and in our real-time garbage collector. We hope that it will used in a variety of different applications, and that it will be ported to most popular compiler and operating system combinations. We welcome contributions of ports to improve the usefulness of this free software.

References

[ABC+83] Malcolm P. Atkinson, Peter J. Bailey, Ken J. Chisholm, W. Paul Cockshott, and Ron Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, December 1983.

[AF94] G. Attardi and T. Flagella. A Customizable Memory Management Framework. In *Proceedings of the USENIX C++ Conference*, Cambridge, Massachussetts, 1994.

[AM95] Malcolm P. Atkinson and Ron Morrison. Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3), 1995.

[BC92] Yves Bekkers and Jacques Cohen, editors. *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992. Springer-Verlag.

[BW88] Hans Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[Coh81] Jacques Cohen. Garbage Collection of Linked Data Structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[Det92] David L. Detlefs. Garbage Collection and Run-Time Typing as a C++ Library. In *USENIX C++ Conference* [USE92].

[Ede92a] Daniel R. Edelson. Precompiling C++ for Garbage Collection. In Bekkers and Cohen [BC92].

[Ede92b] Daniel Ross Edelson. Smart Pointers: They're Smart, But They're Not Pointers. In *USENIX C++ Conference* [USE92], pages 1–19. Technical Report UCSC-CRL-92-27, University of California at Santa Cruz, Baskin Center for Computer Engineering and Information Sciences, June 1992.

[IL90] John A. Interrante and Mark A. Linton. Run-Time Access to Type Information in C++. In *USENIX C++ Conference*, Berkeley, California, 1990. USENIX Association.

[JL96] Richard E. Jones and Rafael D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, August 1996.

[Joh97] Mark S. Johnstone. *Non-Moving Memory Allocation and Real-Time Garbage Collection*. PhD thesis, The University of Texas at Austin, Austin, Texas, December 1997.

[Kak97] Sheetal V. Kakkad. *Address Translation and Storage Management for Persistent Object Stores*. PhD thesis, The University of Texas at Austin, Austin, Texas, December 1997.

[Lip91] Stanley B. Lippman. *C++ Primer*. Addison-Wesley, Reading, Massachusetts, 1991. 2nd Edition.

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An Efficient, Portable Persistent Store. In Antonio Albano and Ron Morrison, editors, *Fifth International Workshop on Persistent Object Systems*, pages 11–33, San Miniato, Italy, September 1992. Springer-Verlag.

[USE92] USENIX Association. *USENIX C++ Conference*, Portland, Oregon, August 1992.

[Wil] Paul R. Wilson. Garbage Collection. *ACM Computing Surveys*. Expanded version of [Wil92]. Draft available at `ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`. To appear.

[16] Luckily, many popular compilers for PC operating systems can emit debugging information in multiple formats; for our system to be compatible with a given compiler, it only needs to understand *one* of those formats.

[17] The type descriptor generator is released under the GNU General Public License (GPL), but the type descriptor records it emits are not covered by the GPL; our system can be (and is) used with commercial applications.

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In Bekkers and Cohen [BC92], pages 1–42.

[WJ93] Paul R. Wilson and Mark S. Johnstone. Truly Real-Time Non-Copying Garbage Collection. In *OOPSLA '93 Workshop on Memory Management and Garbage Collection*, December 1993. Available at `ftp://ftp.cs.utexas.edu/pub/garbage/GC93`.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

[WK92] Paul R. Wilson and Sheetal V. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Addresses on Standard Hardware. In *International Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, September 1992. IEEE Press.