Introspective C++

Hermanpreet Singh

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Computer Science and Applications

Denis Gracanin, Chair
Shawn Bohner
Stephen Edwards

August 20, 2004
Blacksburg, Virginia

Keywords: Metaprogramming, Templates, C++, Introspection

Introspective C++

Hermanpreet Singh

(ABSTRACT)

Introspection has become a significant language feature to enable new component technologies. It enables such capabilities as runtime component discovery, new levels of component flexibility and change tolerance, dynamic reconfiguration and system self healing. Three levels of introspection are discussed: simple type identification, structural introspection, and behavioral introspection

The C++ programming language offers type identification, but neither structural or behavioral introspection. Through its use and combination of several language features, C++ has provided a flexible system for achieving some of the features of introspection without explicitly providing it. Features such as templates, operator overloading, polymorphism, and multiple inheritance have allowed software systems in C++ to build flexible components that tolerate change and support dynamic reconfiguration and self healing. The template system in particular has recently been shown to be more capable than expected, being Turing complete in its own right.

Despite their existing capabilities, the language features have their limits and would benefit from an introspective mechanism. Unlike traditional introspective systems that execute solely at run–time, Introspective C++ has chosen a compile–time approach that tightly integrates with the template mechanism. This approach enables interaction with the other language mechanisms during the compilation, enabling the resolution of many introspective questions before the compiled program is ever run. Furthermore, the mechanism can serve as a base for developing run–time introspective systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Adding introspection to C++ is a delicate operation. The language has a long tradition of efficiency and deep synergy between its various mechanisms. Its template, overloading, operator overloading, and polymorphic mechanisms work together for powerful capabilities [81]. Additionally, the discovery of the template mechanism's Turing completeness [80] has resulted in much new research and discovery in the capabilities of the language [2].

There is plenty of work on traditional mechanisms for introspection [29, 65, 5, 46, 47, 76, 7, 32, 28], but not nearly as much for popular compiled languages [11, 33, 74, 12, 58, 59]. C++ currently has only a base type identification mechanism called Run–Time Type Identification (RTTI). While its combination with the other mechanisms in C++ proves quite useful, it alone is insufficient for building systems using new component technologies like object request brokers [34], query–based debuggers [50], dynamic data components [59], and even the specifications of paths between components [15].

1.1 Introspection

Introspection is a word from psychology: " the direct observation or rumination of one's own heart, mind and/or soul and its processses" . It's been adopted in various forms for programs with some level of self–awareness. In each form, a programmatic interface for analyzing some language structure is presented. The level of analysis varies upon the system presenting the capability. It has been used synonymously with the term *reflection* [69], but that term means the use of introspection in some useful way.

For Java, the definition is "the ability to examine and manipulate a Java class" . The facilities provide mechanisms to determine the name, ancestors, methods, and members of a Java type.

For Perl, introspection is simply access to the interpreter's symbol table . It's similar in

concept for Ruby, defining introspection as to "examine aspects of the program from within the program itself."

But, other systems have a more real–time introspective concept. CORBA's Interceptors let the program know when methods are called on a certain object. Similarly for dynamic proxies in Java. Such introspection isn't based on structure, but *behavior* [4].

For some object–oriented systems, the code is a graph of objects as well. Such a graph can be traversed, allowing introspection of the application's code directly.

The general application of introspection in programming languages is similar to the psychological definition: a program acquires some information about itself. With reflection, it means to use that information to enable better action or knowledge.

1.1.1 Motivations for Introspection

Introspection is a mechanism, but mechanism is irrelevant without use. Introspection has and is used in many systems, with great success that cannot be met without the use of this facility. Such categories of use are described below.

Runtime Discovery

Through the simple introspective ability to get an object's name, a component can assemble a mapping of names to types. Such a mapping allows the component to build a connection with another object through a named, manipulable string parameter. That parameter can come from another component, the network, or a configuration file.

With that, the component can drastically enhance its own capability, through the interactive ability with dynamically–determined partners. Entire graphs of interacting components can be described as a string, which can be stored, computed, and transmitted across a network.

Component Flexibility

Introspection can make components more flexible. By introspection, a component can discover what interfaces are available on an object, and decide what to invoke. The discovery process requires some convention to the meaning behind the introspective data found, such as a naming convention, a meaning to the parameter types and order taken, etc.

Through the combinations of introspection and these agreements, a component can determine the services, capabilities, and/or knowledge of another object dynamically.

Combining this facility with runtime discovery, the component is able to interact with interfaces that need not be known to the original component developer. Components developed

later, by other vendors, or by then–unrelated teams can interact usefully through introspective interface discovery and analysis.

Change Tolerance

Through component flexibility comes greater change tolerance. As interfaces and their implementors change around a component, it can use introspection to avoid dependencies with them. It can go further and validate its requirements of other components by introspecting them, determining if they provide needed services. As software changes, such a self–validating ability has the potential to significantly reduce application breakage during the development process, with the additional benefit of accurate problem reporting.

Dynamic (Re)configuration

As the environment of a system changes, the system can adapt. Through the flexible components, a system can reconfigure its own components to adjust to the environment. The reconfigurations can be validated.

Through dynamic introspection of intercomponent messaging, the system can time and manage the reconfiguration to keep the system consistent before, after, and during the adaptation process. By using a semantic model of the message names, invocation patterns, and participants, the system can ensure a clean transition between configurations at runtime.

Self Healing

Through the introspectively monitoring an object's method invocations, a component could determine that the object has failed or is malfunctioning. For example, if a required sequence pattern of invocations is not followed, or if a method simply takes much too long to finish, a health–monitoring component could determine the object has failed and invoke a system reconfiguration that avoids using it.

Combining the monitoring of and dynamic adaptation to an object's health with the runtime validation of the component graph's semantic integrity, it's possible to build flexible, stable, adaptive architectures. Systems implementing parts of these feature sets have already been implemented using reflection [4]

Granted that this discussion has been filled with high optimism and little mention of the amount of design work required, it is based on a sequence of increasing needs in introspective ability. While introspection won't by itself give much if any of this functionality; it is in some form a prerequisite for all of it.

1.1.2 Current Uses

As listed previously, Introspection can enable some strong capability. In fact, it has. Listed below are a small number of powerful systems developed atop of introspection.

CORBA

OpenORB [4], *The Case for Reflective Middleware* [49], and the technique described in [48] discuss using reflection for CORBA.

OpenORB uses its flexible, reflection–based architecture to enable self–adaptive systems. Using timed automata to describe policies for monitoring, OpenORB has built a framework for monitoring and control for its architecture. Using reflection, the framework can connect to, monitor, and control other components within the system.

Three examples uses are given. First, an adaptive audio stream binding supports adjusting both the buffer size and transmission quality of the audio, depending on the buffer size of the recipient. Access to that buffer size is acquired through the flexible architecture metamodel, based on introspection.

Second, OpenORB is configured as an adaptive mobile middleware platform. It can connect to "any binding type implemented as a configuration of components" [4], allowing it to change between SOAP, IIOP, and other network inter–object protocols. Another framework in OpenORB can use multiple discovery strategies to constantly adapt to the constant change occurring to mobile devices.

Finally, OpenORB has been combined with a component object model (COM) to form OpenCOM. Using OpenCOM and OpenORB, Blair et al [4] are experimenting with a network communications system, integrating all the way down and through the operating system. Using it, they expect to give a network of computers self–healing functionality.

While many of these features have been implemented before, OpenORB has done it all with the same meta–architecture, using reflection at the core.

Kon et al [49], states outright that "Next–generation applications require middleware that can be adapted to changes in the environment and customized to fit into devices, from PDAS and sensors to powerful desktops and multicomputers." Continuing to say that "the reflective middleware model is a principled and efficient way of dealing with highly dynamic environments yet supports development of flexible and adaptable systems and applications." It defines a reflective middleware model essentially as OpenORB has been implemented. Using it, it prescribes that middleware has to be a set of dyanmically–reconfigurable components that can adapt to application and environmental requirements.

Debugging

Introspection is by definition the inspection of a software system. Such inspection is already provided by almost every software development platform, as a debugger. [50] provides a query–based debugger based on SELF [78]. Using introspection for debugging makes perfect sense: it enables an existing requirement to be implemented within the language and runtime, enabling the debugger to access symbolics information with little code of its own. The more powerful the introspective facility, the more of the debugger's traditional workload it can handle.

Debugging using the runtime system prevents bug masking in the debugging process caused by inconsistencies between the introspection and symbolics systems.

In [50], the debugger supports a query language based on SELF. The queries may be saved into a library with the program code, to become a basis for functionality and regression testing. The queries must be side–effect free, but are allowed to call methods on objects. The user is responsible for assuring that the method calls do not alter the state of the recipient, similar to `assert` in C or C++.

JavaBeans

Sun Microsystems provides a system based on Java's reflection called *JavaBeans*. This system defines a semantic model for method naming that allows reflective clients to determine the attributes of a given object. Using the prefixes `set` and `get`, an object identifies its properties by providing methods with one of these prefixes before an attribute name.

For example, an object with a method named `getSize` specifies that it has a property named `size`, and that the object provides read access to it. If another method exists called `setSize`, then the property is externally mutable. The `get` method simply returns the current value. The `set` method takes a single parameter. The type of the returned value from `get` and the parameter for `set` indicates the type of the property. These methods can be simple wrappers around a simple return or assignment statement, or they can be complex methods that calculate or query the value from other data or another data source.

JavaBeans has found success in database–backed applications. The database schema can change during system development, and those changes have the potential of invalidating code using it. To minimize the required code change, a wrapper around the database will map a table row to an instance of what's called an `entity bean`. The entity bean has properties that map to the columns of the table, generally containing one per column. The database wrapper will initialize the entity bean to the values in the row, and can push changes from the bean back to the row. Application code that manages database access can manipulate these beans using reflection, staying independent of their structure (and thusly, changes to that structure).

Application code that becomes invalidated due to properties changing or being removed from the bean is invalidated due to a design necessity; the reason for the change to the database is likely also reason for change to the application logic.

Cocoa

Cocoa [17] is an Objective–C [14] based API originating at NeXTStep corporation as the library AppKit. The name Cocoa is used in its current environment, Apple's Mac OS X. It serves as the newest method to program desktop applications on the Macintosh platform.

It's a flexible system for rapidly developing desktop applications, using introspection in several places to enable that flexibility.

The first and most prevalent use of that introspection is within a tool provided with the development environment: Interface Builder (IB). It provides a graphical environment for building application UI resources. Such resources include the configuration of controls like text boxes, buttons, lists, etc. into containers or windows.

Interface builder uses the introspective facilities of Objective–C and Cocoa extensively to operate. First, its output is a set of archived Cocoa objects, with their interconnections saved as well. Using the built–in serialization mechanism (covered later), all IB needs to do is to be able to instantiate and manipulate the objects, letting the in–library mechanisms do all the I/O work. Similarly, the process to load application resources is trivial: a Cocoa application need only give a single command to deserialize the object graph from a file, getting a reference to it in return. As the Macintosh interface inherently uses multiple windows for a single application [1], the facility is especially useful, as a single resource describing a single document's window can be deserialized once for every open document.

The ability to store and manipulate serialized object references is key to the serialization mechanism. More importantly, it's key to a new level of indirect bindings. A mechanism called Key–Value Coding (KVC) allows the specification of another object, be it a component such as a button or a property such as a button's name, to be given as a path through an object graph. In KVC, a path is specified as a set of names separated by periods. Each name specifies a property to resolve on the current object in the path traversal. The property is analogous to a JavaBean property: while it is given a simple name, it may be accessible by a method following a naming convention, found through introspection. Using introspection, KVC will search the object's method and member list to find something matching its conventions (again, analogous to JavaBean's naming convention). KVC is used extensively in the binding system between web pages and their backing application objects in another Apple software product, WebObjects [16, 15]. WebObjects was rewritten in Java in version 5, and KVC was ported to Java's reflection mechanism. It is still used the same way.

The serialization mechanism in Objective–C is automatic, built–in, and based off of reflection.

All primitive types have built–in serialization, including the string type, `NSString`. During serialization, each object is given a unique identifier and traversed. Each primitive member type is directly serialized; references to other objects are written as those objects' unique identifiers, followed by the invocation of their own serialization. This mechanism allows cycles of objects to be persisted without problems. References to objects that shouldn't be persisted — like library classes which may change between program runs — have proxies written in their place, which refer to the original.

1.1.3 Classes of Introspection

Introspection is available in different systems with different amounts of information available. While each system is slightly different, the levels of information available can be categorized into different clusters.

Even when the system doesn't directly support a type of introspection, a program may implement its own form of it. In each case, the program needs to store its own representation of introspective data, duplicating what's already present in the source base. While it provides additional functionality, it does bring the possibility of incomplete change propagation causing versioning inconsistencies.

Type Identification

The first and simplest form of introspection available is type identification. This mechanism is only available on object–oriented systems that allow inheritance. This mechanism simply allows executing code to check if an object is actually an instance of a descendant of the object's declared type.

For example, if we have a very simple inheritance graph, such as in Figure 1.1, we have two classes, `A` and `B`. The latter inherits from the former. If we have some code that takes a reference to an instance of `A` as a parameter, it can use type identification to determine if that object is actually an instance of `B`. Typically, this mechanism is provided to the language as a runtime operator or keyword. The runtime storage overhead can be minimal; as it can implemented using only the data already provided through the polymorphism mechanism.

This mechanism is so simple and common that it isn't often considered as introspection at all. However, it is a programmatic mechanism to determine an attribute of the program's structure.

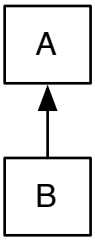


Figure 1.1: Simple Inheritance Graph

Structural Introspection

A common form of introspection that's identified as such is *structural introspection*. In this form, the system presents a programmatic mechanism to analyze the static structure of the program. Even this form has several levels, depending mostly on the organization of the underlying system.

For many object–oriented systems, types can be decomposed. This includes the methods, members, and inheritance graph of a type. Systems that provide this mechanism will often save this information separately from the program text, as preinitialized data. The system runtime provides accessor APIs for retrieving this data as needed. As such, type decomposition requires its own storage overhead for this preserved data.

For "pure" object–oriented systems, the program code itself is represented as a graph of objects. This graph is executed by an interpreter or virtual machine. The graph is available for the program to traverse itself, allowing it access to its entire structure. This includes type identification, type decomposition, and code decomposition. The last feature lets a program look at the executable code, such as loops, math, and method calls. These systems often allow the execution of new object graphs constructed by the program, allowing the program to behaviorally extend itself at runtime.

Interestingly enough, the reuse of the program's code graph means that the introspective ability doesn't directly induce any additional storage overhead; all the data is already available because of the runtime's need for it.

Behavioral Introspection

Another aspect of introspection is of the program's run–time behavior. This form lets a program monitor events occuring as the program runs. Often, this comes in the form of message interception or rerouting. In these cases, each message sent to a given object is intercepted by some intermediary, which can log, manipulate, destroy, and/or forward the message.

Behavioral introspection allows the system to monitor its own actions. Such a facility to

work with messaging allows a program to: through logging, detect bottlenecks; through manipulation or destruction, provide access control; or through forwarding, provide access to remote objects.

Even when the system doesn't provide behavioral introspection, the runtime may use it. It's used in "hotspot" virtual machines [51,78] to detect highly–used portions of code, which may be optimized further or converted to native code. Using the common assumption that 80% of all a program's time is spent in 20% of its code, one can imagine a significant performance boost from a significantly reduced optimization overhead in comparison to more traditional native code generators and optimizers.

Program Implementations

As mentioned earlier, when these facilities aren't available, the program may implement its own.

Type identification can be implemented simply as a type code member in each object that identifies them, provided a general method of getting that member is available. Such a general method could simply take the form of a common base class or a memory–layout convention that enables a utility method to acquire it for any object pointer.

Structural introspection can be implemented with a polymorphic method that returns a set of data structures describing the type. Unfortunately, this data is extremely sensitive to program changes, and is exceedingly difficult to implement in the program without large change–management overhead. As such, external tools may be required to implement this mechanism on a large program.

Code graph traversal is even more difficult to implement and keep consistent than structural introspection. It would be wiser to implement a virtual machine that represents the program text as an object graph, or a precompiler that generates the object graph with the original source text.

Behavioral introspection is often implemented as a particularly flexible messaging mechanism in the machine runtime. If the messaging mechanism isn't flexible enough, proxy classes are often used. A proxy class duplicates the interface of the type it wraps, and is used as if it is the actual object. The proxy can then implement logging, access control, or remote invocation. These proxies can sometimes be implemented without binding to a specific type [13], but often have to be generated by hand or tool [35].

1.1.4 Usage

Each category of introspection listed above will have its own method of access to the program. Each method has its own properties which affect the usability and utility of the introspective

mechanism.

Type Identification

As mentioned earlier, a type identification mechanism will often be implemented as an operator in the program language. It often appears as an infix, keyword–named operation that takes both the object to be tested and the name of the type. The operator can "piggyback" the polymorphism mechanism by introducing a hidden method on the object, returning a type identifier of some sort.

That identifier is often no more than integer or pointer, allowing very fast comparison by the operator. If it's a pointer, it may point to a larger, more in–depth introspective data structure describing the object's type.

C++ provides operators `dynamic_cast` and `typeid` to determine if a type is actually a specific descendant, or to get back a pointer to a structure called `type_info`. The former returns a pointer to the same object as the new type (or `NULL` if it isn't actually a member). If the type is specified by reference, a newly typed reference to the same object, or an exception of type `bad_cast` is thrown.

`type_info` provides a name and operators for comparison and sorting with other `type_info`*s*. No public constructors are provided.

Java provides an operator called `instanceof` which simply returns a `boolean` value if the object is actually the queried type.

Structural Introspection

For structural introspection to work, the compiler must have preserved some description of the type's structure that is available to the program at run–time. Often, this appears simply as static data that run–time libraries can access and traverse. Such a facility usually cannot distinguish between which types to describe and which ones to ignore, causing all but primitive types to have their structural information preserved.

Once that data is preserved, a structural introspection mechanism can provide a pointer back from the type identification system, which provides a starting point for the type information's traversal. Library routines can allow the program to get a listing of methods, members, or base classes, and to traverse their definitions once acquired.

C++ provides no such mechanism. Java provides a runtime mechanism that works essentially as listed above, in `java.lang.reflect` [58].

Behavioral Introspection

Behavioral introspection typically occurs through a messaging infrastructure. That infrastructure can be a simple, mutable dynamic dispatch mechanism, or it can be as complex as an object request broker.

In each case, the system provides an opportunity to intercept messages and act in those messages' place. Objective–C provides this facility built into its own `NSProxy` [13] implementation, which traps attempts to call a method on it, and will pass it to a polymorphic method implemented by subclasses. The subclass can then respond to the method and forward the invocation on to another object.

A project that works on top of Java, called AspectJ [60] uses a concept called Aspect Oriented Programming (AOP) to enable (amongst other things) a mechanism to intercept messages sent to an object. More accurately, it allows work on multiple objects of multiple types to be concentrated into another, which can intercept messages and work on objects as needed. AOP is much bigger than behavioral introspection, and as such the introspective component is provided only as a prerequisite to its intended functionality.

CORBA has its own definition for providing behavioral introspection, on a standard called CORBA Interceptors [36,40]. Combined with the use of a CORBA ORB, interceptors provide a mechanism in C++ to intercept messages sent to a CORBA object.

1.2 C++

C++ is a powerful, extensible, and efficient imperative language for implementing systems. It presents a Simula–based class mechanism atop of C [75], along with a generics mechanism atop of that.

C++'s class mechanism has many features. Beyond traditional polymorphism and inheritance, it allows multiple inheritance, method and operator overloading. This last mechanism allows a C++ class to provide useful meanings for its use in expressions more sophisticated than just message sends. Described below, it also provides a useful interface convention for providing common operations (comparison, copy, etc).

1.2.1 Generics

C++ provides a generics mechanism called *templates*. Templates are a simple mechanism for defining parameterized types and methods by a searchandreplace mechanism. A template is declared with the keyword template, a set of formal parameters, and a named definition. The definition isnt directly usable, it has to be instantiated to a concrete type. Instantiation is the act of using a template with all parameters bound to specific values.

Generics, discussed in more depth later, allow components to work with others that it doesn't need to know about. This enables a certain level of component flexibility. Unfortunately, this requires that the component be completely agnostic to the type it operates on.

For some containers, this works out quite well. For containers that need to do any type of operation directly on the object, such as comparison for sorting, the agnosticism quickly breaks apart. Even worse, the new requirements upon the parameter type aren't expressly written into the component definition in a form that can be verified against. When a container requiring a comparison attempts to do so, it will at the very best fail to compile, or at the worst result in malfunctioning code.

Thankfully, the operator overloading has presented a standard convention for providing such facilities in the contained objects, and a compiler error of "`missing operator<`" serves as a reasonable description of the container's need for a comparator.

1.2.2 RTTI

As mentioned earlier, it does provide run–time type identification (RTTI) [3]. As it "piggybacks" on polymorphism, RTTI is available for any class or structure type that uses that polymorphism. This mitigates the overhead for RTTI for types which would not use it.

The mechanism provides a definition of `operator<` for `type_info`, allowing them to be placed in containers[1]. RTTI allows components the ability to dynamically determine the structure of a graph of objects, but not the structure of any of those objects itself.

1.2.3 Limitations

C++ does *not* provide any introspection further than type identification. No structural introspection is provided in the language. For many applications, uses of the other features of the language suffice. However, for applications that require more, there is little choice: external tools or hand–written descriptions are required.

Unlike Java, external tools for assisting C++ programming have a major hurdle: the C++ syntax. The C++ syntax is so complex that `yacc` or another parser–generator isn't sufficient [75]. Without easily being able to parse the syntax, the tools will unlikely be able to understand the syntactical and type–context of the generated code. Such code will need to be generated type–agnostically, using untyped or poorly typed references where the tool is unable to understand the proper context, hurting the compiler's ability to catch errors.

Hand–written descriptions are an absolute last resort for implementing structural introspection. The costs in programmer time, maintainence, and debugging for inconsistencies will

[1] Actually, proxies to them can be placed in containers, as they are statically allocated and cannot be created at runtime.

make this option prohibitively expensive. It would be preferable to redescribe the types in a separate language that then generates the needed C++ code and metatype information.

1.3 C++ Benefits from Introspection

To avoid the use of external tools or the maintainability problems of hand–written descriptions, introspection could be added to C++. However, such a mechanism has to be carefully implemented. One of the original aims for C++, credited for its success was "that no feature must require really sophisticated compiler or run–time support, that available linkers must be used, and that the code generated would have to be efficient (comparable to C) even initially." [75].

Assuming that a satisfactory implementation can be provided, then C++ has great potential for benefit.

1.3.1 Structural Analysis

Runtime structural analysis would enable a C++ system to build reflective middleware completely within the language (no hand–written or tool–generated descriptions needed), such that it is type–safe and as maintainable as a traditional non–reflective C++ program. In fact, thanks to the advantages of reflection, the middleware and its clients would likely gain additional maintainability advances.

Such analysis would also enable technologies similar to JavaBeans and the Key–Value Coding, Interface Builder, and Serialization of Cocoa. Each needs only runtime structural introspection atop of what C++ already provides to operate.

Finally, as the language and the compilers become more complex, the debugger is constantly trying to adapt [68]. With runtime reflection, the debugger can disentangle itself from much of the change occurring in linker and Application Binary Interface (ABI) formats. Such changes often occur, as in the case of `g++`, where the ABI has changed between subversions (3.1 to 3.2 [77]) of the same compiler! While the runtime libraries of a toolchain get extensive testing from the customers as they test their own code atop of it, debuggers are put under much less stress, and bugs can stay. By reducing the maintenance burden for keeping a debugger up–to–date, the debugger can concentrate on being a more featureful and higher–quality product.

In addition, it becomes possible for third–party vendors (or internal development teams of clients) to develop their own code–inspection tools using introspection and an operating–system API like `ptrace` [67]. These tools can specialize on particular client needs, like data graphing [61] or system querying [50].

1.3.2 Template Parameter Analysis

One unique area where C++ can especially benefit is for template types. As discussed earlier, such types currently have to be as type–agnostic as possible. Such a requirement mandates that template components minimize interaction with their parameters. Any attempt at interaction would add requirements upon the parameter types. Such requirements cannot be represented by any type hierarchy or other error–checking mechanism provided by the language. Only on each attempt to instantiate code from a template type is any error checking done, and the compiler simply tries to check the token stream to see if it can be compiled. Only if the standard C++ errors: type mismatch, undefined method, etc., occur will an error be reported. Otherwise, semantically inconsistent code could be compiled to code that could malfunction at run–time.

If a static introspection mechanism was available, it could be possible to statically verify needed properties of a given type. Furthermore, a template component could adapt, as flexible run–time components have using introspection (see above), to its parameter.

1.3.3 Static Analysis

Unlike other systems, a satisfactory introspective mechanism for C++ would have to integrate with not only the runtime object–oriented mechanisms, but the compile–time mechanisms as well. As mentioned earlier, it has the capability to help templates become more flexible, but it also has the capability to work together with templates to enable further compile–time analysis and optimization.

Templates in C++ have already been used to unroll loops, precalculate trigonometric functions, and eliminate temporary variables [81]. Combined with operator overloading, they have been able to optimize expressions for matrices, eliminating the generation of unnecessary temporaries [79], resulting in code close to if not equivalent to hand–written expressions with multiple matrices. While these techniques are quite powerful as is, they could be further strengthened by allowing the template components to search for optimization paths in the types they operate on.

1.4 Thesis Statement

This thesis describes the implementation and analysis of one attempt to add introspection to C++. Within it is described a language called Introspective C++, which is the language accepted by a modified compiler (based on GNU `g++` 3.3.4).

1.4.1 Goals

Introspective C++ aims to satisfy the needs of the C++ community specifically. This community has requirements which are often different from other language groups, mostly coming from the large variety of applications of the language, and the variety of platforms the language appears on. Introspective C++'s design and implementation is based on several principles:

1. *Impose Nothing Unnecessary* — Due to applications on highly–constrained systems, such as the embedded and real–time markets, some overhead burdens are unacceptable.

2. *Be as Flexible as Possible* — Again, the language is used in many places, and many of these places can use introspection. Due to the demand for introspection in both enterprise systems [59] and the embedded mobile markets [8], the introspective mechanism has to be able to meet many different areas of concern.

3. *Maximize Leveragability* — The potential for introspection's utility is large, so is the reach of C++. An Introspective C++ has to be able to meet the needs of both.

4. *Break no Existing Code* — The C++ language is quite sophisticated [75, 3], and many systems are written in it today. For any addition to C++ to be successful and usable by its user community, it cannot break existing code. As there is no existing introspective mechanism — as we only plan to *add* features to C++ — the only way to break code is to change the meaning of existing constructs. Such constructs include the grammar and libraries in the language.

1.4.2 Strategy

To satisfy our goals, Introspective C++ was implemented as a compile–time mechanism. It added one new type to the library, `std::type_description`. This type is a template that provides introspective information about its parameter. It provides a full structural introspection mechanism in C++.

That template will be filled in with a modified template instantiation mechanism that detects the use of `type_description` and adapts its definition to describe its parameter.

This approach was selected to satisfy all the goals above:

1. *No Unnecessary Overhead* — If it isn't used, it does not impose any compile–time, run–time, or memory overhead to any program using Introspective C++. As seen later, even it is used, it provides complete control over the overhead imposed.

2. *Be as Flexible as Possible* — Templates are the most abstract concept in C++. Using them, Introspective C++ provides as flexible a mechanism as possible that still satisfies the other goals. The use of templates allows the use of the the mechanism at compile time. By instantiating the templates, they can be used at run–time.

3. *Maximize Leveragability* — Through the linkages with the template mechanism, and access to the other features of C++: operator overloading, polymorphism, and multiple inheritance, the introspective mechanism can be used in conjunction with some of C++'s strongest capabilities.

4. *Break no Existing Code* — The standard library's namespace `std` is by definition off–limits to anyone but the C++ standards committee. The name `type_description` is not in the C++ standard at all, much less as a member of `std`. As such, this one addition to the library will not conflict with any existing code that follows the committee's mandate. It is expected that this will be the vast majority of clients.

   As discussed later, Introspective C++ does *not* provide behavioral introspection. Such introspection would require one of two implementation strategies, each with their own problems:

   (a) *Open the Messaging System* — The messaging system would need to be altered to allow message interception and forwarding. While offered in many middleware solutions [26,64,34], such an alteration to the messaging system would encompass a separate project as large as Introspective C++.

   (b) *Enable the Generation of Proxies* — This approach would enable a type to replicate the interface of another. While Introspective C++ does allow a template to peruse the interface, the C++ syntax does not allow for it to duplicate it. The reasons are discussed later under "Future Work."

1.4.3 Improvements over C++

Introspective C++ would enable structural introspection. That introspection would enable the features discussed earlier, including all the functionality of structural, static, and template parameter analysis. Again, it would not enable behavioral introspection. Such a mechanism's implementation in C++ would cause either a doubling of the project's complexity or a violation of the goal to *Break no Existing Code*.

1.5 Thesis Organization

First, we will cover how introspection appears in other languages. This will include languages well–known for their introspective abilities and languages which are related to C++.

Next, we will cover the resulting language Introspective C++, its differences from C++, and its new capabilities. Following that will be a discussion of the implementation of Introspective C++ as a set of modifications to GNU `g++`. With the language covered, we will show some case studies of the language in use.

Finally, we will cover some related work and list our conclusions.

Chapter 2

Introspection in Programming Languages

Introspection is a major capability missing from C++. With it, software components have the ability to simultaneously interact more intimately and more generically.

For example, the JavaBeans [35] architecture uses a combination of introspection and naming conventions to enable the automatic lookup, discovery, instantiation, and initialization of Java classes from a data definition thats only available at class time. One common use is for an application to read in a row of a table in a database and call setY on an entity class (called a Bean in Java terminology) for every column Y in the row. The applications code can use metadata from the database to get all the columns available, and introspect the bean to find and call the appropriate initialization method.

Java, Smalltalk, and ObjectiveC provide introspection via runtime data structures. Such data structures are generated for each class, method, and member. For large systems, the storage overhead can become significant. Also, the introspection requires runtime access to work; adding overhead to the systems runtime.

OpenC++ [11] provides a mechanism for compiler plug–ins to extend the language itself.

2.1 SmallTalk

SmallTalk derives from Simula and Lisp. Originally developed at the Xerox Palo Alto Research Center in 1972, it did not leave the lab until 8 years later as SmallTalk-80. It's based upon the idea that everything is an object, and that all objects communicate via messages.

Simple constructs, like code blocks, integers, strings, conditionals, and looping, are library objects. Through extensive reuse and inheritance, it's easy to leverage the extensive and

highly–integrated existing library objects.

2.1.1 Compilation

SmallTalk comes in more than just a compiler, it's an entire environment. When source text is compiled, the resulting objects are inserted into the environment, ready for immediate use. For specifics, we'll discuss the specifics of one particular SmallTalk environment, Squeak [72].

The environment is saved and restored from an image file, that's bit–for–bit identical across platforms. An application in Squeak is a set of object instances in the environment. More can be instantiated as needed.

More than one image can be loaded at a time and objects can be moved and copied between them.Applications are packaged as images that are copied into new environments as needed. Once copied, they are part of the new environment, and as such, can be invoked as if they were a built–in feature of that environment.

2.1.2 Syntax

SmallTalk's syntax is very simple: a way of specifying messages sent to objects. Such message send actions can be grouped together into blocks, which themselves can respond to messages. The syntax specifies three ways to send messages to objects: unary, binary, and keyword.

Unary messages have a simple format: *recipient* `message`. The recipient indicates which object to be sent the message, and the latter is the message to be sent. Binary messages are almost as simple: *recipient message parameter*. The first two are the same as before, but the last value, the parameter, is given with the message to the recipient.

Keyword messages are similar to method invocations in Java or C++. The syntax is similar to the others: *recipient* [*namesegment: parameter*]+. The message would have a name like `setMax:min:`, with parameters following each of the colons.

To force precedence, parenthesis are used. These are necessary to send messages to objects returned from other messages. For example: `array at:1 set:5.` would send the message `at:set:` to `array`, while `(array at:1) set:5.` would send the message `at:` to `array`, and the object returned would then receive the message `set:`.

Variables have no static type in SmallTalk. Variables are all references to objects, and all objects are typed. Without static type, variable declarations are simply statements of their name in the proper contexts, such as member, variable, and parameter declarations.

Conditionals are very simple: relational operators return objects of type `Boolean`, which has the method `ifTrue:ifFalse:`. It takes two parameters, code blocks that run, depending on the value of the boolean value. Loops work similarly: a block of code is an object that has

a method called `whileTrue:`, which takes a block as a parameter. That parameter block is run repeatedly as long as the block's code results in `true`.

2.1.3 Type System

All data are within objects. Arithmetic types are thus also objects, which implement methods for mathematical operations. Precedence for operations is very simple: left to right, except for parenthesis.

There is no "native" array concept in SmallTalk [76]; instead, a `Array` object exists, that uses methods `at:` and `at:put:` to read and write its values. Two dimensional arrays are implemented with class `TwoDArray`, and higher dimensions are only possible with embedding `Array`s within each other.

As SmallTalk doesn't give type to its variables or method parameters, no static type checking can be performed. Only when an object receives a message it doesn't implement or a method throws an exception is an error flagged. In fact, the former will actually just throw an exception of type `doesNotUnderstand` .

2.1.4 Object Model

Objects have a type, members, and methods. The type is another object by itself, called the *Class Object*[1], referred to by a global variable with the type's name. To subclass a type, send that type's class object a `subclass:instanceVariableNames:classVariableNames:-poolDictionaries:category:` message, which creates a new class object with the added instance, class, and pool variables.

To construct an instance, call `new` on the class object. The return value is a new instance, which can then be used as necessary. A garbage collector will automatically destroy any unused instances lying around.

Operators are methods themselves, with non–alphanumeric names. The expression $1 + 2$ leads to the instance of `Integer` for 1 receiving the message `+` with a parameter of the instance of `Integer` for 2.

There isn't a heavily–enforced access control mechanism like in C++ or Java. Instead, methods are placed into named categories, and some categories have the prefix `private`, which the runtime and compiler do not treat specially. The prefix `private` simply means that the interfaces and implementations may change in a later version, so it's best not to use them.

Inheritance and polymorphism exist in SmallTalk. Single inheritance only [32], without the

[1] [7] extended this concept further into *explicit metaclasses* [28], resulting in the Classtalk platform

concept of formal interfaces as in Java[2]. Polymorphism is ubiquitous; to override a method, simply define it in the subclass. There is no special keyword to mark a method as overridable like `virtual` in C++, nor is there a method to prevent override like `final` in Java. Without access control or specially marked polymorphic members in SmallTalk, the language presents no obstacle to arbitrary changes of a type through subclassing.

SmallTalk has no support for generics. As variables are untyped, it's understandable to avoid the issue.

2.1.5 Runtime

The SmallTalk system is a full environment that contains all objects and interacts with the user. Programs in SmallTalk exist within *images*, virtual machine memory segments that can be added together. Objects are fully persisted within the images, and they live until garbage collected.

The SmallTalk libraries are extensive and easy to use. The language has a built–in documentation system, where every object and method can have a string associated with them, which can then be browsed. A programmer can use an object browser to peruse the extensive library of collections, utilities, graphics, and network classes.

SmallTalk does provide introspective abilities through `ClassDescription`. It's possible to retrieve all method names, categories, and members through its APIs. As there is no access control, all data is available to every client.

2.2 Objective-C

Objective–C [14] is a SmallTalk–derived object–oriented layer added to standard C. Originally developed by Brad Cox and the StepStone Corporation [25], it was licensed by NeXT and then reimplemented by Dennis Glatting, Richard Stallman, and Kresten Krab Thorup. The last implementation has been the official GNU runtime since 1993 [25]. Currently, it's used as the vendor–preferred API for new application development on Mac OS X. The primary compiler is the Free Software Foundation GNU Compiler Collection (GCC).

Objective–C provides a dynamic runtime environment running on the native processor; no virtual machine is used. Although there is no garbage collection, reference counting is built into the libraries, and the burden on the programmer is minimal to interoperate with it. Furthermore, the runtime provides interesting functionality, such as the ability to extend a class at compile, link, or run time. Finally, the runtime interacts with the client code to enable additional capabilities, discussed later.

[2]Without typed variables, such a mechanism would not be useful anyways

Objective–C has been used for years for next–generation systems like NeXTStep and WebObjects, and has always enjoyed good tool support. Apple provides almost all support for Objective–C today. The APIs have changed names, from an `NX` prefix to `NS`. Today, both GNUStep [31] and Apple's implementations use the new prefix.

2.2.1 Compilation

Objective–C compiles like C or C++: a set of source files (with the `.m` suffix) are compiled to object files. The source and object files have no required naming relation to the code within. The compiler stores additional class metadata describing the methods, members, and layout of the aggregates defined.

The code first goes through a preprocessor like C. The Objective–C preprocessor is nearly identical to C's, except for an additional `#import` directive, which works like `#include` with built–in multiple include guards.

Executables generated from Objective–C are traditional in nature; `ELF`, `COFF`, and `Mach-O` are supported in modern toolchains. NeXTStep and Mac OS X package executables into directories called bundles, which are treated as a single file within the user interface.

Like executables, libraries are also generated like any other on the platform: `DLL`, `.so`, or `.dylib`. NeXTStep and Mac OS X package libraries into directories called frameworks, again treated as single files within the user interface. Application bundles and frameworks can contain each other as needed.

While Objective–C provides no direct application management infrastructure, its cohorts NeXTStep and Mac OS X, through their bundling and framework mechanisms, allow drag & drop application installation and removal.

2.2.2 Syntax

Objective–C's syntax is essentially C with a modified version of SmallTalk put on top. Like C++, a header file declares the type's messages and members. The keyword `@interface` specifies the name of the type, its supertype, and any protocols it implements.

The primitive types are 100% C. Instances of objects aren't allowed on the stack, only pointers. All of C's declaration syntax is supported, and pointers to objects are declared as they would be in C++. Objective–C also supports typeless object types called `id`. It's actually a typedef to `void*`, but is allowable everywhere an object pointer would otherwise be required.

The syntax for loops, conditionals, and free functions is identical to C. This allows conditionals to check pointer values directly, without a redundant `==0` like Java. Even though

free function definitions and calls to them are identical to C, message sending and method definition is very different. Those are based on SmallTalk.

2.2.3 Type System

As mentioned earlier, all of C's primitives are supported. Structures are also supported. Wrappers for them exist, although fewer in number than Java. `NSNumber` covers all numeric types. `NSString` and `NSMutableString` cover strings. Through them, one can put primitive types inside containers and use them as regular objects.

Beyond that, Objective–C has two interrelated constructs, the `@interface` and the `@implem-entation`. The former declares the members and the list of methods visible to clients. The latter contains all the method implementations. Note that additional methods can be defined in the `@implementation`, which will be added to the method list as any other. These additional methods can be called by any client due to the dynamic messaging system.

C's arrays are immediately available. Furthermore, `NSArray` and `NSMutableArray` provide object wrappers around fixed arrays. Objective–C's containers all provide useful abstractions upon all their contents, and these two types enable them for arrays. Arrays of arrays are allowed in C's traditional arrays and in Objective–C. `NSArray` and `NSMutableArray` allow them as well by recursive embedding.

Objective–C's can operate with weak or semi–weak type checking. The original libraries and runtime use the type `id` to refer to any object. Any message could be sent to it, and any type is convertible to `id`. Internally, `id` is a `typedef` to `void*`. However, users complained about the lack of static error checking, so some basic capabilities were added.

A pointer can be declared to any `@interface`, and messages sent to that pointer are checked against the `@interface`'s declared API (not the set of messages the `@implementation` defines). Any messages sent to the object, but not declared by its API, are flagged as warnings by the compiler. Because the object could easily respond to many more messages than it declares, a compilation–stopping error would be inappropriate.

For primitive types, the type checking is exactly the same as standard C. Traditional `(type)` casts easily remove any hindrances provided by the type system.

Objective–C's object model is modeled after SmallTalk. Only single inheritance is supported, but with two other important features: categories and protocols.

Categories were originally designed to let developers split up the `@implementation`s of types across several source files. They allow additional methods to be added to an existing type. Furthermore, a category's implementation of a method will *override* the original. Note that this isn't the same as overriding a method through inheritance; the category's methods are part of the original type. The overridden method is never used. If two or more categories implement the same method, then the selection of the method implementation that runs

depends upon the load order of the categories' respective object definitions at run–time.

Protocols work essentially like `interface`s in Java: a set of methods that are declared but undefined. Classes declare their conformance to one or more protocols and implement their methods. Due to the fairly untyped nature of Objective–C, the actual declaration of conformance is mostly for documentation reasons. Just as often as protocols are used, classes will declare *informal protocols*. Informal protocols exist only in documentation as a set of methods a class may choose to implement. Objective–C's runtime allows a class's implemented method list to be checked, so a client can see if it implements a method from an informal protocol before sending it the appropriate message.

At the heart of Objective–C's runtime is the `Class`. The `Class` contains the full description of a class: its superclass, name, version, size, instance variables, method list, and conformed protocols. The `Class` is a `typedef` to `struct objc_class*`, a C structure. While the definition of `objc_class` is available through the header files, several C and Objective–C functions and methods are available to the programmer for abstracted access.

The `Class` contains the list of methods implemented by the type. All method names are hashed to an opaque `struct objc_selector*`, generally just called a *selector*.

Every instance of the same name, no matter which interface or implementation it exists in, is hashed to the same value. This way, message names have no binding to any particular class. Using this property, one can send any message to any type.

Upon compilation, the compiler links to the runtime in two ways. First, the `Class` structures are all defined and filled. Second, every message send has the method name converted to a selector, and then passed to the runtime function `objc_msgSend`.

`objc_msgSend` searches the recipient's method list for the selector, and dereferences the listed pointer to the proper code. The method list is actually a hash table for speed. If the object doesn't implement the method, all of its ancestors are searched. If none of them implement it, then it sends a "second chance" invocation, by sending `forwardInvocation:` to the original recipient.

`forwardInvocation:` allows an object to handle any message it wants. The name implies the most common use for this feature: to let an object act as a proxy for another. An object can implement `forwardInvocation:` as a quick check upon its proxied object to see if it responds to the parameter, and if so, sends the message onwards. `forwardInvocation:` has a default implementation in the root class `NSObject` that throws an exception and logs the failed message call.

Objective–C doesn't provide any type of parameter overloading at all. In fact, parameter checking isn't a strong feature at all in Objective–C. Only when the optional static type checking features are used can parameter checks be done at all.

A message is dispatched upon its selector, which is based only on the message name. When static type checking isn't used, the return type is assumed by the runtime to be of type `id`.

Any client sending a message to a type that returns anything else has to cast the value.

As mentioned earlier, a form of overloading occurs with categories that define methods with the same name as one in the original type. This is a dangerous practice, as another category could do the same and ambiguate the selection process.

No operator overloading is provided in Objective–C.

Objective–C only allows access control upon members. Although one can "hide" methods by defining them in the `@implementation` without declaration in the `@interface`, instances will still respond to messages of that name from any sender.

Members can still be `@public`, `@protected`, or `@private`. The syntax for accessing them is identical to accessing a structure member via a pointer. For further control, the traditional C methods of defining pointers to undefined (such as `objc_selector`) or unlisted types (`void*`) are still useful.

As mentioned earlier, Objective–C allows a type to inherit from one other. All the member variables and methods are inherited. `@private` members are part of the subclass's definition but are inaccessible. All methods are inherited and accessible. Any method can be overridden, as there is no concept of a `final` or non-virtual method in Objective–C.

While upcasting is directly allowed, it's not very useful. C allows arbitrary casting and Objective–C doesn't really need a type's name to send it a message. Except for cases of voluntary static type checking, inheritance in Objective–C is essentially useful as implementation inheritance only.

Objective–C provides the most flexible polymorphism of any language described here. It has a binding mechanism much more dynamic than most OO languages. Furthermore, it allows the *client* much more control over an object's behavior than most other languages: categories and dynamic message proxies let the client modify the interface and behavior of the type. Only `@private` data members aren't accessible to client code via subclassing or categorization.

Exceptions only appeared in their current form in Apple's compiler in Mac OS X 10.3. An earlier form existed that was based on macros, but it won't be described here. The current syntax is very similar to C++: a `@try` block encloses code that may `@throw`, followed by one or more `@catch` blocks, each with a typed parameter. Like Java, an optional `@finally` block can follow, with code that will always be run.

Only subclasses of `NSException` may be thrown. The `@throw` construct begins a scan and unwind process on the stack for a `@catch` block that will handle the exception type or one of its ancestors. Any `@finally` blocks are run along the way.

Objective—C doesn't have a compile–time generics system. Its dynamic messaging system helps mitigate the loss. For example, the library containers all support the method `makeObjectsPerformSelector:`, which sends a message with the parameter selector to ev-

ery contained element. Furthermore, the weak and optional static typing mechanism reduces the value of a generics facility at all.

2.2.4 Runtime

As covered earlier, the runtime plays a large part of the Objective–C system. At startup, all the loaded classes' `Class` descriptions are initialized. From there, the standard C entry point `main` is run.

Memory is best described as "semiautomatic." Its inherently reference–counted, with the appropriate support infrastructure directly within `NSObject`. Two methods, `retain` and `release`, increase and decrease a reference count in the recipient object. When that count reaches zero, the object calls `dealloc` on itself, and releases any resources it holds. `dealloc` will then call `release` on any member objects it has, causing the proper cascading effect.

For additional functionality, Objective–C supports an `autorelease` message. `autorelease` will add the receiver to the current autorelease pool, an instance of `NSAutoreleasePool`, a container. `NSAutoreleasePool` takes ownership of the object. When the pool is itself `release`ed, it will call `release` on all of its contents.

`NSAutoreleasePool`s can be nested within each other, and are automatically supported by the GUI library in Objective–C, `AppKit` (now called Cocoa [17]). The most common use is within the event loop: an autorelease pool is created when an event enters the application, and `release`ed when the event loop reenters it waiting state. In the meantime, any `autorelease`d object will have a convenient lifetime that lasts just long enough to be used for handling the event. Any object that needs to live longer can be `retain`ed by any other object, which will hold the only reference count to it when the event loop reenters the waiting state.

While not as large as Java's, Objective–C's library is certainly quite powerful. More importantly, it's much more flexible thanks to good leverage of the runtime's dynamism. For example, the serialization mechanism directly allows connections between objects to be serialized easily, allowing a GUI builder to directly connect data objects to GUI components without necessitating the generation of code or other glue. Furthermore, serialization works for nontree structures, thanks the the runtime's ability to directly analyze the objects.

Of particular interest in the library is the single–threaded nature of it all. The entire system was designed to run within a single thread, with a messaging bridge to connect threads together. The messaging bridge works exactly the same as a full interprocess communication system, in fact identically to its distributed objects mechanism. That will be described below.

Most of the runtime's abilities come from two places: the messaging system and functionality built into `NSObject`. `NSObject` provides the following APIs:

1. `conformsToProtocol:` — If the object exports a specific protocol's API.
2. `respondsToSelector:` — If the object responds to a specific method.
3. `isKindOfClasss:` — If the object has a specific class as its type or in its type inheritance hierarchy.
4. `isMemberOfClass:` — If the object is of a specific type.

These methods constitute around 90% of what's needed by most applications. The remainder go and directly analyze the `objc_class` structure for more information. For the most part, the dynamic messaging infrastructure and weak typing reduce the need for direct introspection in the system.

2.2.5 Case Study: Distributed Objects

There need not be any "what if" scenarios here; Objective–C's basic runtime libraries provide distributed object (DO) functionality. In fact, this functionality's used for more than interprocess communication; Objective–C uses its DO mechanism for inter-thread communications; avoiding the traditional complexities of synchronization.

The distributed object system works quite simply: a generic proxy class for the transport mechanism masquerades as the type it's proxying. The proxy tells its clients that it responds to all the selectors of its destination. Any messages sent to it are serialized and sent through the transport to the recipient. The system is so simple because of both the flexible messaging system and the built–in serialization system.

The serialization of messages is significant; such a serialized form could be replicated, replayed, or even analyzed and restructured. For a DO system, another question arises, when should a parameter be serialized and when should it be proxied? If the parameter is a simple, flat data structure, it can and should be serialized. However, if it's a complex object with relationships to other objects, it should be proxied. Serializing and restoring such an object will certainly be complex, possibly lossy. Such a determination also can't be made by the compiler; these are design–level decisions.

For that reason, additional keywords exist in Objective–C to tag parameters that should be proxied. The terms `byref` and `bycopy` indicate which parameters are sent by reference and copy, respectively

2.2.6 Case Study: Serialization

Serialization is built into Objective–C as well; it's a prerequisite for marshaling a message. There are two parts to serialization: wrappers for each primitive type and a protocol for

serialization. The latter, `NSCoding` lets serializing objects interoperate and cooperate for more complex types. The wrappers implement the protocol and take most of the grunt work out of serialization.

As Objective–C doesn't allow the iteration of an object's members, there is no support for automating the serialization process of a complex type; some code has to be written. In practice, the process is often much simpler than it sounds. Objective–C's runtime library is full of high–quality containers that will serialize themselves and all their contents. Often, it's advantageous to use one or two containers instead of many members; letting them do most of the serialization work.

2.3 Java

Java [45] is a programming language derived from Objective–C and C++. Developed at Sun Microsystems, it has become extremely popular for World Wide Web applications. Originally intended as an embedded systems language, one of Java's original goals was platform independence. That goal has lead to a toolchain and runtime that runs on the vast majority of the worlds' desktop computers and some of the larger embedded platforms. Linux, Windows, MacOS, Symbian, PalmOS, and many variants of Unix are just a few of the operating systems Java runs on today.

Due to its ease of access, strong APIs, and simple programming model, Java has become one of the dominant languages used today. Java's virtual machine (VM) executes Java Byte Code, an assembly–like language atop of a stack–based virtual processor. However, the virtual machine has some unusual intelligence to it: specifically that it knows about objects. Java is thoroughly object–oriented, and that shows in the virtual machine's architecture.

The virtual machine starts and loads up classes from a `ClassLoader`, an object responsible for loading class definitions into the system for use. `ClassLoader`s can be custom–written and added to the system to work in unison with the default file–based loader, allowing networked access or on–the–fly class generation [44].

Beyond knowing how to load segments of code in classes, Java's virtual machine has another capability granted from its knowledge of the object model: garbage collection. The Java language is fully and transparently garbage collected. Unreferenced objects, even those who keep cycles of references to each other, are properly destroyed and their memory reclaimed during the normal execution of a Java program.

Modern virtual machines are used in high–load, mission–critical environments. They have become very solid and their performance has been extensively optimized. Today's VMs watch for often–used sequences of Java bytecode and compile them into native machine instructions for further optimization. Also, completely–native compilers exist for Java [63] that eliminate the need for a virtual machine and its overhead.

2.3.1 Compilation

Before a class can be loaded, it has to be compiled. Compilation occurs before program execution. Each class in Java has its own `.java` file that is compiled into a `.class` file. Each `.java` file can only have one publicly–accessible class in it, defined with the same name as the filename. All static checks are performed, inside a class and between classes. Because classes are compiled into separately–loaded files, additional checks are needed at runtime to make sure the file is compatible with the system.

In general, run–time compatibility errors show up as accesses to undefined or inaccessible members or methods of a class. These failed access attempts exist as exceptions, which can be caught. Due to the need to catch these kinds of errors, and other reasons, Java saves a rich amount of information about each compiled class directly within the `.class` file, which is loaded and maintained by the virtual machine during execution. Java's introspection mechanism exposes this information to the programmer.

Even though there isn't a true linking stage within the Java compilation model, there's still a need to distribute single files for entire applications or libraries. The Java utility `jar` acts almost exactly like the standard Unix `tar` utility; only that it uses the `.zip` file format and stores a manifest file inside the archive. The Java virtual machine's standard `ClassLoader` can directly read `jar`s and load class code from them.

2.3.2 Syntax

Java's syntax for basic procedural use is very similar to C and C++. However, it cannot be used without using the object–oriented features. As implied by the compilation process, Java only compiles classes. All code must exist within classes, either in methods, constructors, or static initialization blocks.

To demonstrate, take a look at Figure 2.1. Like C, the entry point for any program in Java is `main()`. However, Java doesn't allow any code to exist outside of a class definition, hence the need for the otherwise–useless `MainContainer`.

The syntax for declarations, loops, and conditionals is nearly identical to C and C++. The standard set of `for`, `while`, and `do` loops exists. Two main differences exist: (1) statements given as parameters to conditionals must be of `boolean` type, and (2) there are no explicit references or pointers. For example, the comparison `(i==0)` could be written as simply `(i)` in C or C++; if all the bits of `i` are zero, then `i` is considered `false`. Also, the variable `mc` is a reference to a heap–allocated `MainContainer`. Unlike C++, no `*` or `&` is necessary.

For primitive data types, the declarations are nearly identical to C and C++. Every other declaration is implicitly a reference to an object. References to references are not allowed. Although the references keep the same lifetime as primitive types, the lifetimes of the objects are different. All objects live on the heap through a call to `new`, and die upon garbage

collection.

```
public class MainContainer {
  int loop = 32;

  public static void main (String args[]) {
    MainContainer mc = new MainContainer ();
  for (int i=0; i<loop; i++) {
    System.out.println ("Hello World");
    if (i == 0) {
      System.out.print ("!-");
    }
  }
  }
}
```

Figure 2.1: Simple Java Program

2.3.3 Type System

Java has taken a pick–and–choose approach to its type system from C++ and Objective–C. The primitive types come from the latter. Primitive types like `int`, `float`, `double`, `boolean`, and `char` all exist in Java and behave as they do in C, C++, or Objective–C. The only difference is that their specific sizes and precisions are completely defined; unlike Objective–C or C++ where it varies upon the specific processor architecture.

Similar to Objective–C's `NSNumber`, Java has fully–fledged class types that peer the primitives. `Integer`, `Float`, `Double`, `Boolean`, and `String` all provide wrappers around a primitive value, as well as comparison and conversion operations. As discussed later, Java's containers can only contain full objects, and these peers allow the primitive types to be used.

Java doesn't have the concept of an explicit pointer or reference type; instead, every variable declared to be of a class's type is really a reference to an instance allocated on the heap. Every such variable has to be initialized with a call to `new`, and any other variable set to be of the same value will refer to the same object. While references to objects can be declared `final` like primitive types, making them immutable, the referred objects are always mutable. To simulate a const object as in C++, the traditional Java method to return immutable objects is to define a subset of the object's interface with only "getter" methods that allow the query but not modification of an object's state. The true object's exposed interface is a superset of this, and it formally "implements" it (discussed more later).

Arrays in Java are essentially unidimensional. They hold either primitive types or references to objects. However, arrays are also objects, and thus references to arrays can be stored in arrays as well. Through this double-indirect mechanism, multidimensional arrays are implemented in Java. The immediate benefit of the flexibility is clear: arrays are simple to understand and flexible to use. For example, the references to objects could be polymorphic; allowing further dynamism.

Arrays, like other Java objects, are always mutable and garbage collected. They also "know" their length, and can be queried for them as needed. Arrays also have a peer, `Array`, which provides similar wrapping facilities as the other peers.

Java uses a similar type system as C++. All variables are declared to have some type, either primitive or object. Attempts to assign the variables values of other, incompatible types are errors. Downcasting from a class to one of its subclass types is allowed. However, there is little allowance for implicit conversion; only upcasting. Even narrowing conversions between floating point types from literals are errors!

2.3.4 Objects

Java's object model is a mix between C++ and Objective–C. The methods are declared and used almost identically to C++ syntax. However, the inheritance and polymorphic mechanisms within Java function more similarly to Objective–C.

Messages in Java look and act almost identically to C++: essentially functions with a hidden `this` pointer back to the object. An attempt to call a method not implemented by the message recipient is flagged as a compile–time error. Furthermore, methods can be overloaded: more than one method can be declared and defined with the same name, as long as their parameter list differs.

Unlike C++, default values for parameters, nor the overloading of operators are allowed. There is one exception: Java's `String` class has concatenation operators defined, with definition for all the primitive types and behavior to call `toString` on all other objects given as parameters.

Class types fall into three categories: classes, abstract classes, and interfaces. Classes have all methods defined, member variables, and are constructible. They can also inherit from one other class, and *implement* any number of interfaces. Interfaces only have methods declared without implementation. Classes implement interfaces by implementing all of their methods.

Abstract classes are unconstructable objects with one or more methods unimplemented and marked `abstract`. Unlike interfaces, they can have some methods defined for subclass use, but take the role as the only superclass. Subclasses must implement all abstract methods of their abstract superclass and any implemented interfaces to be constructable.

For classes that have non-memory resources allocated, a *finalizer* can be defined, which is

run when the object is garbage collected. However, the specific time of execution, or even a guarantee of execution, isn't provided.

Access control is extremely C++-like: private, public, and package access is allowed. Access sections like C++ aren't provided; each member and method has to have its own qualifier listed, otherwise it's assumed to be package–level access. Inheritance and interface implementation, however, are always public.

Objects can be queried of their type through several ways. First, the `instanceof` operator returns a boolean value specifying if the object is an instance of a specific type or subtype thereof. Next, `Class` provides the comparison methods `isInstance` and `isAssignableFrom`, both of which compare compatibility with another object. Most often, the `instanceof` operator is used.

As mentioned before, classes can inherit from exactly one superclass. When one isn't mentioned, it's Java's standard `Object`. As a consequence, every object in the Java system inherits from Object. Without multiple inheritance, questions about diamond inheritance, ambiguous superclass references, and the like are completely avoided. Also as mentioned before, each compiled class has a description that's loaded, maintained, and checked by the virtual machine. Such a description is available to the developer as an instance of type `Class`, available through `Object`'s `getClass` method.

All methods are implicitly polymorphic; the C++ keyword `virtual` is assumed. The Java keyword `final` will specify a method that cannot be overridden in base classes. For a method to be overridden, it has to have the same access level as the original, and the same signature (method name and parameter types). As upcasting is an implicit conversion, Java objects are often treated as if they were instances of their base class or an implemented interface. Method calls to are routed to the closest ancestor's implementation.

Exceptions are based on C++: a `try` block containing code that may throw, one or more `catch` blocks that handle a specific type of exception, and specific to Java, a `finally` block for cleanup code that runs even if a stack unwind is in progress.

Unlike C++, Java's exceptions only use class types: throwing an integer or floating point value isn't possible. The virtual machine is a source of many exceptions as well. Illegal actions, such as trying to call a method on a null reference or casting an object not of the specified type, are trapped by the VM and result in exceptions being thrown in the running program. This gives the running program a reasonable chance to trap the error and continue execution.

Java has a generics system in its 5.0 beta as of August 2004 [55]. Java's compiler has a simple preprocessor that allows the declaration of generic types. Generic types in Java have a similar syntax as C++: angle brackets denote parameters to the generic type. The parameters are used to denote types used for method parameters, member types, and return types. The compiler will flag attempts to use an instantiation of a generic type that doesn't match its definition. For example, a generic container will not allow insertions of objects that aren't

instances or subclasses of its parameter.

Behind the scenes, the generic types have only one instantiation that's shared between all uses. The parameter type names are all converted to `Object`, and compiled as a normal Java class. This way, the traditional one–to–one mapping of a `.java` source file and the compiled `.class` still exists. Unfortunately, the only gains from the generics feature are some type safety and reduced need for casting; none of the more powerful capabilities generics provide in C++ are available in Java.

2.3.5 Runtime

As mentioned before, the virtual machine loads Java `.class` files via a `ClassLoader`, which returns a `Class` object to the VM. At startup, the VM is given a single class name to load, which must have a `public`, `static` method named `main` taking a single parameter: an array of `String`. That method is run with the command line options given at the VM's invocation, with VM–specific options removed.

`main` may spawn any number of threads, which are supported natively by the VM. Included with the ability to create new threads are in–language synchronization abilities, such as the ability to make a method `synchronized`: callable only from one thread at a time. The virtual machine provides the threading and enforces the synchronization, even if the underlying platform doesn't do it natively.

Like C, C++, and Objective–C, the program lives only as long as `main` runs, even if other threads are still active when `main` completes. During that lifetime, all memory allocated is tracked and managed by the VM. Objects and graphs thereof with no incoming references are garbage collected and their memory reclaimed.

One of Java's most powerful assets is the wealth of standard libraries. The standard library contains nearly 2,000 classes and interfaces in nearly 100 packages. Together, a platform–independent system for developing desktop, web, and command–line applications exists, with facilities for almost every common development need.

Due to the easy packaging and distribution of platform–independent code, Java also has one of the richest 3rd party library communities.

Within the standard libraries lie Java's introspection mechanism. The mechanism's libraries allow programmatic access to the class information the VM maintains. As hinted before, the `Class` type provides the key interface for accessing a type's information. With it, members, methods, interface, and superclass information is all available. Moreover, `Class` provides a query function for getting the appropriate `Class` instance for a type with a specific name. Such an ability, connected with the `ClassLoader` mechanism, allows a Java program to assimilate code that was not available at the original system's compilation, loading it, linking it, and running it when appropriate.

`Class` provides a healthy API, we present only the relevant subset for this discussion. All the `public` constructors, methods, and members are accessible via `getConstructor`, `getMethod` and `getField`. Plural versions of these methods exist that return arrays of each as well. These methods return `Constructor`, `Method`, and `Field` objects.

`Constructor` is essentially a factory class. Given the parameters it needs for initialization, its `newInstance` will return a newly constructed instance of the type. It allows querying of the required parameter types via `getParameter-Types`, which returns an array of `Class` objects. Note that primitive types do have `Class` objects, but they must be passed to `Constructor` as wrapped objects.

`Method` acts almost as a selector in Objective–C. Similar to `Constructor`, it has a `getParameterTypes` method for accessing the parameters. It also has `getReturnType` and `getName` for getting the full method description. `invoke` in `Method` takes a recipient object and a set of parameters and invokes the method on the recipient.

`Field` provides `get` and `set` methods for objects and pairs of these methods for each primitive type. All of them take a recipient object which contains the member variable in question. The primitive type pairs are named as `getInt` and `setInt` which return and take primitive types for primitively–typed members.

All three classes, `Constructor`, `Method`, and `Field` will throw exceptions when they are used inappropriately. Examples include passing the wrong parameter types, sending to the wrong object, or passing invalid values. Because the classes must have type–agnostic interfaces, these errors cannot be caught by the compiler.

2.3.6 Case Study: Serialization

Java provides built–in serialization [54, 38, 6]. By implementing a zero–method interface `Serializable`, an object can be serialized. The primitive types can also be serialized.

Use `ObjectOutputStream` [56], a wrapper around a normal Java `OutputStream`, to serialize the object. `ObjectOutputStream`'s `writeObject` method will serialize the object to the stream, and its peer `ObjectInputStream`'s `readObject` will deserialize it.

By sitting atop of the standard stream mechanism in Java, serialization works atop of any byte–stream I/O mechanism. Included in the standard libraries are files and sockets. The developer may write their own stream classes and send serialized objects over them with little difficulty.

Serialization is almost completely transparent to the object being marshaled. The only time a class need worry about serialization is when it's got members that don't implement `Serializable`. While most of Java's classes do, some don't for obvious reasons, like `Thread`. For these members, the class must mark them `transient` in their declaration.

Furthermore, the class may need to know when it's being serialized or deserialized, so that it can adjust its state. For example, it would have to reconstruct any `transient` members that were lost during serialization. The class can define private `readObject` and/or `writeObject` methods which will be called during the relevant processes. From there, it can prepare for serialization or fully restore from it.

Another option in Java is the `Externalizable` mechanism, which does less of the work by itself, in exchange for greater control of the serialization format.

2.3.7 Case Study: Distributed Objects

Java provides a basic distributed objects mechanism called Remote Method Invocation (RMI) [57]. By implementing a zero–method interface `Remote`, an object specifies that it can be remotely invoked.

When a message is sent to a remote object, the parameters are serialized. Those parameters which implement `Remote` are given remotely–accessible identifiers, which are sent in their place. A parameter that implements neither `Remote` nor `Serializable` can't be sent. From there, the virtual machines interact to transport and dispatch the message.

2.4 OpenC++

OpenC++ [11, 9, 10] is an extension upon C++ that allows metaclasses to be defined, that plug into the compiler. It's been used as the basis for a query–based debugger [50, 39]. These metaclasses define a translation layer between the input source code (in an extended C++ called simply OpenC++) and the C++ language proper. The metaclass system allows full awareness of the structure of defined types (introspection) and of the context of each use of the defined types. The metaclasses can modify both as the source code is compiled.

OpenC++ is a general–use system. A similar approach [43] has been used specifically for the high–performance computing arena.

2.4.1 Compilation

The compilation system is dynamic: the input source code is compiled into plugins for the compiler, which then monitor and modify the compilation of itself. Currently, only a single stage is allowed: the input source code is OpenC++, and the metaclasses must emit standard C++. An extension is planned to allow metaclasses to emit OpenC++, which is then fed into additional passes through the metaclasses for them to emit C++.

The metaclasses have two parts. First a `TypeInfo` object is generated for every declared

type, describing its methods, members, and inheritance hierarchy. Second, a metaclass is its own OpenC++ type derived from `Class`, containing event handlers that plug into the compiler. The handlers are called when the compiler encounters the type itself or any uses of it. The handlers are fed `PTree`s: constructs similar to Lisp S-expressions that describe the parse tree of the code being compiled. The handler can modify the `PTree` before returning it to the compiler. The handlers use the `TypeInfo` as needed to determine the necessary changes to the `PTree`.

2.4.2 Introspective Abilities

`TypeInfo` stores the traditional C++ structural definition of a type: its members, methods, and inheritance hierarchy. As such, it provides all the information we need for an introspective mechanism. However, it's only available for use to metaclasses in OpenC++ and runtime code in C++; there's no integration or availability to the template mechanisms within C++.

To connect the introspective data to the template mechanism in C++; one has to write metaclasses that generate type traits describing the information found in the respective `TypeInfo`.

2.4.3 Case Study: Distributed Objects

OpenC++ allows a simple implementation for the basic network distribution of objects. A metaclass can simply handle calls to the type's methods, inserting any desired marshaling and I/O code necessary in the call's place. Alternatively, a metaclass could generate a full stub class to act as a local proxy for the remote object.

In OpenC++, a programmer can define new keywords that automatically specify the metaclass. For example, the programmer could register a keyword `distribute` to specify a `DistributedObject` metaclass. A simple "hello world" class is shown in Figure 2.2.

```
distribute class Greeter {
public:
    std::string getMessage ();
};
```

Figure 2.2: OpenC++ Class with Distribution Keywords

`Greeter`'s metaclass would be `DistributedObject`. `DistributedObject` would make `getMessage virtual`, create a subclass called `Greeter_proxy_ala_DistributedObject`, and generate any glue necessary for the runtime distributed messaging system to let it create and configure the proxy.

2.4.4 Case Study: Serialization

OpenC++ makes easy work of serializing a type. The metaclass can add new methods to the type to serialize and deserialize it. These methods can serialize as much as they can, pushing off any other parts to run–time code.

A type will typically have several categories of members to serialize:

1. *Simple Members* — Primitive types that can be directly serialized. Such as integers, floats, and C strings.
2. *Irrelevant Members* — Members that need not be serialized. For example, caches.
3. *Simple References* — Pointers to other objects completely owned by its single container, but in need of run–time support to serialize. For example, a dynamically–allocated array.
4. *Complex References* — Pointers to objects that may already have been serialized, or may transitively refer back to the container. For example, a graph of objects with cycles in it. In such cases, a placeholder token may be needed to refer to an object already serialized.

For each type, a keyword can be added. For example, the class in Figure 2.3 has members of each type.

The metaclass would have to keep track of which `DataObject`s were already serialized, and simply refer to them as needed. For the simple references, the metaclass's serialization methods would call `serialize_array` to bring the data values in and out of the memory buffer to the parameter member references (`val_string_length` and `val_string_body`).

```
persistent class DataObject {
    // simple members
    int a, b;

    // irrelevant members
    transient int c,d;

    // simple references
    single (serialize_array) int string_length;
    single (serialize_array) char * string_body;

    // complex references
    shared DataObject *parent;

    size_t serialize_array (mode_t inorout,
                            memory_buffer &buffer,
                            int &val_string_length,
                            char *&val_string_body);
};
```

Figure 2.3: OpenC++ Class with Serialization Keywords

Chapter 3

Introspective C++

By adding an introspective mechanism to C++, we can receive some if not all the benefits listed in Section 1.3. As mentioned earlier, getting such a mechanism correct entails much careful design.

3.1 Language Integration

Introspective C++ is a superset of C++, and as such has to integrate with C++. To implement such integration, we have to remember the goals:

1. Impose Nothing Unnecessary
2. Be as Flexible as Possible
3. Maximize Leveragability
4. Break no Existing Code.

Upon analysis of these goals, several additional constraints became obvious for C++:

1. *Add no New Syntax* — The mechanism must work within C++'s existing syntactical standards. While it's possible to add some new syntax construct to enable Introspective C++ that would not otherwise compile (therefore preventing a violation of goal (4)), we have several demotivating factors:

   (a) *C++'s Syntax is Complex Enough As Is* — C++'s syntax is so complex that it essentially requires a hand–written, recursive descent parser [75]. On the programmer side, the syntax is sufficiently complex that any additions will likely cause hesitation to use the mechanism.

(b) *The Existing Mechanisms are Pretty Powerful* — The existing language has a powerful synergy between its mechanisms, developed over time through the work of many people. It's likely that any additions to the syntax would be severely overclassed by the existing language mechanisms.

(c) *The Interactions are Hard to Predict* — With the prior two factors already in place, a natural consequence is that any new syntax would likely have unintentional and equally likely, undesirable interactions with other parts of the C++ syntax.

For these reasons, we decided it best to avoid syntax extensions altogether.

2. *Interact With Existing Mechanisms* — As mentioned earlier, C++ has quite a few existing mechanisms in place. These mechanisms can be made to interact favorably. Any metadata layer presented in Introspective C++ should easily integrate with these mechanisms. One particularly important mechanism to consider is the template mechanism, which is strictly compile–time in nature.

3. *Avoid a Run–Time Reflection Mechanism* — This constraint is particularly important. To interact with the template mechanism, we can't make the introspective mechanism a pure run–time system. Furthermore, a run–time system would require that metadata be stored in the executable or some run–time accessible resource. Such a requirement breaks our first goal: Impose Nothing Unnecessary. Our chosen solution shows a method that doesn't need run–time storage, hence proving that run–time storage isn't always necessary.

With these constraints in place, we have a strong motivation to attempt to reuse an existing language mechanism to implement the Introspective C++ language.

3.1.1 Template Use

The chosen strategy is to provide a new template that takes the type to introspect as a parameter. This introspective type is inside the namespace `std`. To invoke it on a type `T`, that is, to introspect `T`, client code need only refer to a type named `std::type_description<T>`.

While the template syntax this strategy requires (shown later) is less imperative in nature than the rest of C++, hence less likely to be known by the majority of programmers, the use of a template satisfies our goals and constraints:

1. *Impose Nothing Unnecessary* — Template definitions are created on demand for each use. Furthermore, these instantiations can have a size of zero[1]. Finally, a template's

[1] Unless it's directly constructed; if a subclass of a template is constructed, the template base class can have a size of zero [3].

definition is fully accessible to the compiler's optimizer. If a template has `inline` definitions, those definitions can be propagated up to the calling context, allowing for minimal or zero overhead for use.

2. *Be as Flexible as Possible* — Combined with the derived constraints, the use of an introspective template provides the only satisfactory use of a language mechanism. As such, it is the only mechanism that can solve the clause "as possible." Despite the lack of alternatives, it provides a flexible solution. Templates can be used for static analysis *and* for the generation of run–time data and code.

3. *Maximize Leveragability* — By using a template, each introspective instantiation is its own type. As such, it can be discriminated by the template system and used with good leverage. No other language mechanism allows the creation of new types without hand–coding.

4. *Break no Existing Code* — As the namespace `std` is off limits for client code and other vendors, any names added to `std` not defined in the standard are available for use without fear of conflict.

5. *Add no New Syntax* — A template's syntax already takes in a type as a parameter. It requires no additional syntax.

6. *Interact with Existing Mechanisms* — As template instantiations are distinct types, they are accessible to the template, polymorphic, inheritance, and overloading mechanisms already in C++.

7. *Avoid a Run–Time Reflection Mechanism* — The mechanism is compile–time in nature.

Types Only

Our template is defined to take only one type parameter. As such, it cannot take variables, other templates, nor a function. The variable constraint isn't a great hindrance — all variables in C++ have a type.

Introspective C++ can't dirctly introspect a template: such an ability would require that the declaration for `std::type_description` take template–template parameters, which would break compatibility with normal types. However, an instantiated template, one with all parameters filled, is a normal type and thusly can be introspected through `std::type_description`.

Public Data Only

Even though C++ specifies that "it is access to members and base classes that is controlled, not their visibility." [3], we have chosen to prevent any introspection of nonpublic data. That

restriction has several motivations:

1. *Single Definition* — Only a single definition of the introspective data is possible in the language (only one definition for each instantiation is allowed). By providing only public data, a single definition can be used by any client without concern for access violations.

2. *Better Encapsulation* — Private and protected data is considered implementation–specific, and is not the concern of external code.

3. *Simpler Use* — As it's reasonable to assume that component writers mostly do not want to break encapsulation, they would most often want to avoid introspecting private or protected data. If all of it were available, each client would have to write more complex logic to avoid it.

3.1.2 Language Features

To help understand Introspective C++, it's useful to review some key language mechanisms from C++. These mechanisms are powerful and are critical to using Introspective C++ effectively.

Templates

C++ provides a generics mechanism called *templates*. Templates are a simple mechanism for defining parameterized types and methods by a search–and–replace mechanism. A template is declared with the keyword `template`, a set of formal parameters, and a named definition. The definition isn't directly usable, it has to be *instantiated* to a concrete type. Instantiation is the act of using a template with all its parameters bound to specific values.

Figure 3.1 gives an example template of a C++ class. `M` has a single data member, whose type is parameterized. `M` has a constructor to initialize it. `M` is instantiated below the definition with a parameter `int`. At this point, the compiler will create a new type `M<int>`, whose definition looks like Figure 3.2. That definition only exists within the compiler's memory; it's immediately compiled into object code without ever being visible to the user.

The parameters can either be other types or values of primitive types. Default values and other templates (called template–template parameters) can be used as well. Figure 3.3 gives an example. The parameter `I` is an integer, suitable for an array size specification. The parameter `U` is another template, which must take `T` as a parameter. Finally, `N` is an integer parameter, which if left unspecified (that is, if `M` is instantiated with only three parameters), then `M` will assume that `N=0`.

```
// the template class M
template<class T> class M {
    T value;
public:
    M (T v) { value = v; }
};

// an instantiation of M
M<int>  m(3);
// m.value is now 3.
```

Figure 3.1: C++ Example Template

```
class M<int> {
    int value;
public:
    M (int v) { value = v; }
};
```

Figure 3.2: C++ Instantiated Template

```
template< class T,    // type parameter
          int I,      // integer parameter
          class U<T>, // template template parameter
          int N = 0>  // default value
class M {
    T[I] values; // an array containing I instances of T.

public:
    M (int v) { value = v; }
};
```

Figure 3.3: C++ Complex Template Parameters

Specialization

Templates can be made for class types and methods within them. Top level class type templates can be *specialized*: redefined for specific parameter values. A specialized template has one or more parameters fixed to specific values, and a new definition attached. The most common use is to optimize a template for a specific parameter or class of parameters. For example, Figure 3.4 specializes `M` for pointer types. All pointer types have exactly the same

definition by converting them to `void*`, which is legal for all pointer types. The benefit of this specialization is that only a single definition is defined for all `M` with pointer parameters, reducing the total number of instantiations of `M`. Such a reduction can significantly reduce object code size in a large application.

```
template<class T> class M<T*> {
    void * value;
public:
    M (void * v) { value = v; }
};

template<> class M<char*> {
char * buffer;
public:
    M (char *c) {
     buffer = malloc (strlen(c)+1);
     strcpy(buffer,c);
    }
    ~M () { free(buffer); }
};
```

Figure 3.4: C++ Specialized Templates

The first definition does not conflict with the original definition of `M`, rather it redefines `M` when `T` is some pointer type. Note that it still takes a parameter `T`, but specifies that this definition is for instantiations of `M` when a pointer to `T` (`T*`) is given as a parameter. As `T` is still a template parameter, this allows the definition to work for any case when `M` is given a pointer to some type as a template parameter.

The second definition recognizes a C string and creates its own dynamic array to hold its own copy of the string. The default implementation given earlier and the definition for pointers, would have simply copied the pointer, causing `M<char*>` to simply store a reference to the original string, instead of having its own copy. The specialization rules in C++ will give preference to `M<char*>` over `M<T*>` as possible, as the former is a more specialized definition — specifically, that `M<char*>` is a full specialization while `M<T*>` is a partial one.

3.1.3 Techniques

To use a template–based introspective mechanism, one has to be familiar with a set of techniques that, used together, is called *metaprogramming*. Alexandrescu [2] provides in–depth discussion on the techniques and potential applications. Metaprogramming has been

analyzed and used extensively within the C++ community [80, 52, 37].

To understand how to use all that Introspective C++ provides, it's essential to know these techniques.

Type Traits

C++ allows a template to have a different definition depending on one of its parameter types. Such an ability is called template specialization, and when at least one of the parameters isn't specialized, it's called partial template specialization.

This mechanism allows one to build a compile–time mapping between one type and another. If one defines a template differently for every parameter value of interest, and leaves no definition for the general case, that template has a one–to–one mapping with its parameter types. Figure 3.5 has an example.

```
template<class T> struct is_serializable;
template<> struct is_serializable<int> {
    enum { value = 1 };
};
template<> struct is_serializable<std::ostream> {
    enum { value = 0 };
};
```

Figure 3.5: A Type Trait

`is_serializable` has no standard (default) definition, but it does have definitions when `T=int` or `T=std:ostream` When client code tries to evaluate `is_serializable<int>::value`, it will get 1 in return. Similarly, for `T=std::ostream`, `value=0`. An attempt to evaluate `is_serializable` for something other than `int` or `std::ostream` will result in a compile error, as there's no default definition.

Another template can use such a value as in Figure 3.6. In this example, the type `serializer` defines an array of integers called `validate_serializable`. That array's size is determined by the `::value` of `is_serializable`. Specifically, if `::value` is 1, then the array size is defined as 1. If it's zero, the array size is defined as -1, which is illegal in C++ and results in a compile–time error. If there's no definition of `is_serializable` for `T`, then the compiler doesn't know what `::value` is, and gives a compile–time error.

Here, `serializer` will fail to compile for values of `T` that either have `is_serializable` undefined or with `value` set to zero.

```
template<class T> struct serializer {
    int validate_serializable[is_serializable<t>::value? 1:-1];
};
```

Figure 3.6: Example Use of a Type Trait

Recursive Templates

Specialization also lets us define recursive templates. All that's necessary is to define a template that refers to another instantiation of itself, and specialize one of these instantiations so that it doesn't refer to another.

Recursive templates have to recurse over something. Alexandrescu [2] covers the concept of a *typelist*. A simple type definition is used recursively, much like a `cons` in Lisp.

```
template<class H, class T> struct Tuple {
    typedef H head;
    typedef T tail;
};

struct NullType {};

typedef Tuple< int, Tuple< char, NullType > > shortlist;
```

Figure 3.7: A Small Typelist

Figure 3.7 provides that simple type definition as `Tuple`. A terminator for that list is given as `NullType`. Finally, an example `Tuple` is given in `shortlist`. `shortlist` provides a typelist of (`int`, `char`, `NullType`). The last value denotes the end of the list.

A simple example will show how to use such a type. In Figure 3.8, we will take a typelist and create a structure containing one element for each type specified in the typelist.

In the first two definitions, we use inheritance to aggregate definitions of `Container` with the different parts of the typelist. The second definition terminates the recursion.

The next two definitions, for `ContainType`, traverse the recursion of `Container`, as a utility to the last definition. That final definition, for `get`, returns value `N` from a `Container<C>`.

Template Lists

Template lists are simply templates that have numeric parameters that represent particular elements in a list. To traverse a template list, we use a form of recursion as the typelist ex-

```
template<class T> struct Container
  : public Container< typename T::tail > {
  typedef T list_type;
  typedef typename T::head value_type
  typename T::head  value;
};

template<> struct Container<NullType> {};

//--

template<class C, int N> struct ContainType {
    typedef ContainID<
        typename C::typenamelist_type::tail, N-1
      >
      result;
};

template<class C> struct ContainType<C,0> {
    typedef C result;
};

//--

typedef<class C, int N> struct get {
    typedef ContainType<C,N> ret_type;
    typename ret_type::value_type& operator () (
            Container<C>& c
        ) {
            return static_cast<ret_type&>(c).value;
    }
}
```

Figure 3.8: Recursive Use of a Typelist

ample above, only in a simpler, integer–only form. A traditional factorial example illustrates the syntax in Figure 3.9.

`struct factorial` gives the factorial for its parameter. Seeing how it does it requires a bit of explanation. The default definition of `factorial` is recursive. For a parameter `N`, it attempts to evaluate the result of `factorial<N-1>::value`. To prevent infinite recursion,

```
template<int N> struct factorial {
  enum { value = N * factorial<N-1>::value };
};

// Redefined factorial for N=0.
template<> struct factorial<0> {
  enum { value = 1 };
};
```

Figure 3.9: Compile–Time Factorial Example

we define another version of `factorial` for `N=0`. As mentioned before, C++'s syntax is getting pushed beyond it's original intent. The examples push it further.

It will help to understand how `factorial` works by seeing it in action. First, let's define a shorthand: `<n>`, which is short for `factorial<n>`, where `n` is some constant integer. Imagine `<3>`. `<3>::value` is defined as `3 * <2>::value`, which is defined as `2 * <1>::value`, which is defined as `1 * <0>::value`. `<0>` has a different definition, namely `<0>::value = 1`. The compiler will substitute in the definitions as used to resolve the original case of `<3>`. It'll result in a definition for `<3>::value = 3 * 2 * 1 * 1`. The optimizer's constant–folding pass will convert that to `<3>::value = 6`.

Reifying Templates

Sometimes it's necessary to save a value retrieved from compile time, into a run–time structure. The primary difficulties come from building a structure that can be perused by run–time code, without knowing the specific type of the data structure being traversed.

Simple inheritance assists greatly. We define a base type that contains the data required. Then, we define a template subtype with its own constructor, which initializes the base class data. This subtype can have a member for another step in a recursion, allowing it to assemble lists of descriptions.

Figure 3.10 shows the technique in action. All it does is store a linked list of decreasing integers. The first definition provides a base type that client code can use. The next provides a recursive definition, which defines as a member the next step of the recursion. The final provides a terminating case.

The second definition builds a linked list of instances between instances of `base_init`. Client code need only receive a pointer to an instance of `base_init`, declared as `base*`. Then, it can simply traverse the `next` field to iterate through the list. The final definition sets its `next` pointer to zero (`NULL`).

```
struct base{
    int value;
    base *next;
}

template<int N> struct base_init : public base {
    base_init<N-1> next_init;
    base_init () : value(N), next(&next_init) {}
};

template<> struct base_init<1> : public base {
    base_init () : value(1), next(0) {}
};
```

Figure 3.10: A Runtime Representation of Templated Data

3.2 Type Identification

C++ already provides type identification at run–time and compile–time. At run–time, any polymorphic type will respond to the `typeid` operator, as well as enable use of the `dynamic_cast` operator. The former returns a type called `std::type_info`, which is displayed in Figure 3.11[2].

`type_info` provides a constant C string for the name, but the contents of that string are vendor–dependent, and allowed to be empty [3]. Through the two operator overloads (`operator==` and `operator!=`), it allows comparison between `type_info` instances. Through `before`, it allows sorting. However, the private copy constructor `type_info(const type_info& rhs)` and `operator=` prevent any construction of new objects, even copies. As such, it mandates that only references to statically–allocated (by the compiler) can be stored in a container.

At compile–time, types are distinct and distinguishable throughout the language mechanisms. Type checking verifies compatibility and produces diagnostics. Function, method and operator overloading provides different semantics for different types. Template specializations allow different template definitions, depending on the parameter type.

With both run and compile times covered, Introspective C++ doesn't need to any any functionality. As it simply adds a template in `std`, it doesn't interfere with the existing language mechanisms for type identification.

[2]Comments, inline method definitions, `#if`s, and the "internal interface" section removed for brevity. The code listed is covered under the GNU General Public License, which requries that I note that it has no warranty.

```
class type_info
{
  public:
    virtual ~type_info();

  private:
    type_info& operator=(const type_info&);
    type_info(const type_info&);

  protected:
    const char *__name;
    explicit type_info(const char *__n);

  public:
    const char* name() const;

    bool before(const type_info& __arg) const;
    bool operator==(const type_info& __arg) const;
    bool operator!=(const type_info& __arg) const;
};
```

Figure 3.11: `std::type_info`, from `g++'s <typeinfo>`

Introspective C++ provides additional information above that of `type_info`. By nature of the used language mechanism, this information is only available at compile–time. A technique for reifying that information for run–time use is given in Section 3.1.3.

3.2.1 Provided Type Traits

Introspective C++ provides some information in a form similar to the *type traits* mentioned earlier. For any parameter type `T`, Introspective C++'s `type_description<T>` provides enumerations and `typedefs` with useful values.

All the enumerations mentioned are provided for every type. To indicate a "true" value, they use the traditional C convention of being equal to anything but zero. A value of zero indicates "false." This convention only applies to enumerations that indicate boolean conditions. All such enumerations have a `is_` prefix on their name.

Some enumerations serve a double purpose: to indicate if a class of items exists, and if so, how many. For these, a prefix of `num_` is provided. To indicate that the class of items does not exist, the value of the enumeration will be zero. Any other value indicates the number of items that do exist.

Classification

`type_description` has several enumerations that classify the parameter. Together, they cover all possible parameter types that can be input into `type_description`, and as such there will always only be one that indicates a true condition.

The enumerations are named after the classifications they indicate, and all have the `is_` prefix:

1. `is_class` — Is this a class type?
2. `is_union` — Is this a union type?
3. `is_enum` — Is this an enumeration?
4. `is_function` — Is this a function type?

Arrays of each type are handled slightly differently, and are mentioned below.

Arrays

An array is simply another type, repeated multiple times. This "inner type" can be another array. To introspect arrays, Introspective C++ provides an enumeration `array_length`, which indicates the number of elements in this array type. If `array_length=0`, then this type is not an array. Although this is just like a `num_` enumeration mentioned above, the term "length" is significant to arrays, and a zero–element array already has useful semantics in C++.

If `array_length` is nonzero, then the type being repeated is indicated in a `typedef` called `inner_type`. Such a `typedef` is only present in this condition. That inner type may be another array type if the parameter was a multidimensional array.

Functions

Functions cannot directly be given to `type_description` as a parameter. However, the function's signature (with return type), can be given as a parameter. In such cases, `is_function` is nonzero and a `typedef` called `return_type` indicates the type returned from a function of this type.

The types of the parameters may be retrieved through the structural introspection mechanism.

Aggregate and Primitive Types

For aggregate and primitive types, one additional enumeration is provided. This enumeration indicates whether or not the compiler considers this type to be "POD". POD [3] is a term defined as "Plain Old Data." It signifies a type which has semantics similar to a C structure. POD types may be primitives or aggregates (structures, unions, or classes) that have:

1. *Default Implementations of the Default Constructor, Copy Constructor, Destructor, and* `operator=` — If unspecified by the programmer, the compiler creates these methods automatically for every aggregate type. These methods assume that a bit–for–bit copy of an object will successfully result in a complete, consistent, and independent object being created. This works well for primitive types and C `struct`*s*, but fails when any initialization or allocation/destruction needs to take place.

   If a type has a member that is not POD, then the compiler must generate appropriate methods that call the member's customized methods. In such a case the containing type is not considered POD, as the default bit–for–bit semantics may have been broken.

2. *No Polymorphism* — The type cannot be POD if it has any virtual methods or a virtual destructor.

POD types have the advantage of allowing their handling without calling methods on the type. By giving away some of the control over their handling, they can be manipulated very generically, often using generic, optimized code.

Common Elements

Every type given to `type_description` has two values specified: `align` and `name`.

`align` indicates the *alignment* requirements of the type. That is, the type's address in memory must be a multiple of `align`. This requirement usually stems from a processor requirement for memory access. Keeping proper alignment is required on some machines, and simply imposes a significant penalty hit on others.

`name` is the declared name of the type. If the parameter type is actually a `typedef`, `name` is the name of the actual type, not the `typedef`'d name. This follows the definition in the ANSI standard: "A *typedef–name* is thus a synonym for another type. A *typedef–name* does not introduce a new type..." [3]

As `name` is a string value, it can't be given for compile–time use[3]. Instead, `name` is declared as a static member of `type_description`. If `name` is never used, it's never instantiated, thus never costing run–time storage overhead.

[3] C++ has no primitive for a string, thus providing no mechanism for a template to use it

3.3 Structural Introspection

The largest additions that make up Introspective C++ are in the structural introspection mechanism. In this mechanism, additional introspective subtypes are added as templates *inside* `type_description`. All of these inner templates work in a similar fashion.

All of these templates describe a single item in a list. The list may be the list of base classes or the list of members. In each case, the containing template indicates how many items there are, with an enumeration prefixed by `num_`. The inner template takes in an integer parameter, ranging from zero to the enumeration minus one.

The indexing mechanism is consistent with the C++ semantics for array access. An attempt to reference a template with an out–of–bounds index will result in the instantiation of an empty template. Such a reference by itself will succeed, but any references to any values within it will fail. This is a limitation of the C++ template mechanism.

3.3.1 Methods, Constructors, Destructors, Operators

For `struct` or `class` types, it's possible to retrieve the list of methods, constructors, destructors, and operators defined. As they all have the same syntax and similar semantics in C++ — similar to functions — they are all represented with the same mechanism.

Access

That mechanism is the enumeration in `type_description` called `num_methods`. Despite its name, this value counts not only methods, but the constructors, destructors, and operators within a type definition. Note that overloads of the same item, including constructors, are not counted separately here; those are counted within an inner overload introspection mechanism, described below.

The inner template of `type_description` describing each of these items is called `method`. `method` takes a single integer parameter to index which item is being discussed.

Provided Information

Each item's description has some enumerations, a `typedef`, a pointer, and a static variable.

Classifiers: Enumerations identify the type of item being described:

1. `is_ctor` — Is this item the constructor?

2. `is_dtor` — Is this item the destructor?

3. `is_operator` — Is this item an overloaded operator?

   Distinguishing which operator this overloads is currently unavailable at compile–time. Only through analysis of the `name` is discrimination possible.

Attributes: Attributes about an item are also provided. Note that even if the attribute isn't possible for the classification of item (e.g. a virtual constructor or static destructor), the attribute is still provided, with a value of false.

The attributes correspond directly to keywords defined in the language. The meanings of those keywords can be found in the ANSI standard [3].

1. `is_const` — Is it declared as `const`?

2. `is_virtual` — Is it declared as `virtual`?

3. `is_static` — Is it declared as `static`?

4. `is_nothrow` — Is it declared to `throw ()`?

Description: Finally, three more elements of information are provided: a `type`, a pointer (`ptr`) to the item, and a static member containing the `name` of the item.

`type` is the function type of the item. It can be used for further introspection to retrieve the parameter list or return type.

`ptr` provides a pointer to the actual implementation. It's declared (inside the compiler's additions) as statically–initialized to the address, so accesses to it can be optimized out if the compiler can determine which method to call.

`name` provides a C string of the method's name. No name–mangling [27] occurs. For constructors, this will simply be the name of the type. For destructors, the type name will have a prefixing "$\tilde{}$". Method names will be listed verbatim.

3.3.2 Overloads

When a constructor, operator, or method has overloads, they are represented in Introspective C++ as an inner type within `method`. This inner type is indexed the same way as `method`: an enumeration in the containing type (`method`) called `num_overloads` specifies the number (or zero if the item is not overloaded).

When overloads are present, everything but `name` from member moves into `overload` — because everything but name may differ between overloads. To access a given overload `N`

of method M in type T, use the template std::type_description<T>:: template method<M>:: overload<N>. The keyword template has be placed in the middle to allow the parser to follow along [3].

3.3.3 Members

For any member of a struct, union, or class type, std::type_description::member provides member introspection. Similar to method, an enumeration num_members provides an upper bound for indexing an inner template member.

member provides the following information:

1. type — The type of the member. This is a typedef that itself can be used for introspection.
2. name — The name of the member. A static C string (char*) giving the declared identifier name for the member.
3. is_static — An enumeration identifying static members.
4. ptr — A pointer to the member.

   Care has to be taken with the relationship between ptr and is_static. When is_static is nonzero, ptr will point directly to the single, global instance of the member. When is_static is zero, ptr will be a C++ pointer–to–member [3, 27, 53], which must be dereferenced against an instance of the type.

3.3.4 Base Classes

For any struct or class that inherits from another, base_class provides information. The access mechanism is the same as for methods or members: an enumeration (num_bases) indicates the upper bound for an inner template (type_description::base_class) indexed by an integer parameter.

base_class provides only two pieces of information. First, an enumeration is_virtual to indicate if the type is a virtual base [3]. Second, a typedef named type, identifying that base. Similarly to the other typedef*s*, this one can be fed back into the introspective mechanism for further analysis.

3.3.5 Function Parameters

For function types, a very simple integer–indexed inner template param, whose upper bound is specified by enumeration num_parameters, gives only one piece of information: a typedef

named type, indicating the type of the parameter taken at that place in the argument list. That type can be introspected as well.

3.3.6 Template Parameters

If the type given to type_description is actually an instantiated template, Introspective C++ provides a mechanism to peruse those parameters.

The inner template is named template_param. Its index bound is specified by num_templ-_parameters.

template_param is slightly different from the previously–mentioned inner templates. The first thing provided is an enumerated value param_type. This value is always equal to one of two enumerated type values in type_description: tp_typename or tp_int[4]. The former indicates that the template was originally declared to take a class or typename parameter. The latter indicates an integer (int) parameter.

```
template<class T, int N> struct templ {
T array_of_ts[N];
};

std::type_description<templ>::
  template templ_param<0>::param_type
is
  std::type_description<templ>::tp_typename.

std::type_description<templ>::
  template templ_param<1>::param_type
is
  std::type_description<templ>::tp_int.
```

Figure 3.12: A Template Taking Typename and Integer Parameters

For example, the template in Figure 3.12 has two template parameters: T and N. The former is a typename, the latter an int.

[4]Aliases exist for each, specifically tp_class for the former and tp_enum for the latter.

Chapter 4

Implementation

To implement Introspective C++, a compiler for it must be made. C++ is a very complex and sophisticated language, and implementing a compiler good enough to support the language features needed to use Introspective C++ (templates, specialization, etc.) would be prohibitively complex. As there are several compilers that already support C++ well enough for our needs, it is far more feasible to modify one of them to support Introspective C++.

4.1 Compiler Evaluation

Having selected to modify an existing compiler, we proceeded to evaluate and select one to modify. Several options existed: a commercial compiler such as Microsoft Visual C++ [23], Comeau C++ [18], or Metrowerks CodeWarrior [20], an open–sourced compiler like a OpenWatcom [42], or the venerable GNU `g++` [73, 62].

The key criteria for evaluation were support for needed C++ language features and the ability to extend the compiler to add on the new Introspective C++ features. The language features needed included templates, template specialization, partial template specialization, and good support for nested template declarations (e.g. `std::type_description<T>::template method<N>`).

4.1.1 Microsoft Visual C++

Microsoft Corporation sells and supports a C++ development platform for their Windows platform. In the newest release as of August 2004, the latest version, Visual C++ .NET 2003, supported significantly more of the C++ standard; most importantly, partial template specialization [24].

While the compiler does now have the feature set to support the C++ features needed to effectively use Introspective C++, it does *not* support the addition of new language features through any form of language or compiler extension mechanism. That conclusion was made after investigating what `was` available in that area [22], and noticing that language extensions was not a part of it.

4.1.2 Comeau C++

Comeau Computing's C++ compiler is well known for its in–depth track of the C++ standard and its quick implementation within the language. It claims full support of [18]:

1. Support for all core language features of C++03.
2. Complete implementation of export
3. Addresses all issues which have been identified as defects in Standard C++ as per "TC1"

These features were more than enough to satisfy our language support criteria. However, the compiler did not present any form of extension mechanism we could find in the freely–available documentation. However, when testing our implementation of Introspective C++, we used Comeau's free online compiler as a check to verify the C++ syntax of various declarations and structures.

4.1.3 Metrowerks CodeWarrior

Through our own use of the compiler on other projects, we determined that Metrowerks CodeWarrior supported the needed language features of C++. It also supports a plug–in mechanism. Unfortunately, that mechanism only supports the ability for the integrated development environment to invoke other compilers and related tools. The C++ compiler itself did not support any language extension mechanism.

4.1.4 OpenWatcom

Sybase's Watcom compiler was known for producing good code for the Intel compilers, being used for id Software's original DOOM. Unfortunately, the product was not actively maintained for a very long time, eventually being open sourced into OpenWatcom [42].

OpenWatcom, being open–source, would have allowed us to implement our language extensions. However, due to the age of the compiler, we felt it too risky to attempt modifying

it only to find a large deficiency in C++ compliance later. Such a risk was deemed quite possible, as the compiler was deemed "no longer commercially viable" [41] after their last version of 11.0b. The `README.TXT` file for Version 11.0b was dated February 24, 1998. The first edition of the ANSI C++ standard, was dated September 1st, 1998.

4.1.5 GNU `g++`

The GNU project's `g++` compiler received significant improvements as Linux became popular, being the only compiler capable of compiling the kernel and libraries, due to their use of compiler–specific extensions [19]. Again, through own experience on other projects, we had found that `g++` had the language features necessary for Introspective C++.

As `g++` is both open–source and had the necessary C++ features already in place, we selected it as the starting point for developing a compiler for Introspective C++.

4.2 The Gnu Compiler Collection

The Free Software Foundation's GNU Compiler Collection (GCC [73]) is the basis of software development on all major open-source platforms, including Linux, Mac OS X's Darwin core, BeOS, FreeBSD, NetBSD, and OpenBSD. Supporting dozens of platforms, GCC compiles C, C++, Objective C, Fortran and Java. The collection's based on a single source tree of mostly-shared portable C code.

Today, GCC is a mature compiler set with strong support for the our language of focus, C++. The GCC C++ compiler, `g++`, has very good coverage of and compliance to the Ansi C++ Standard [3].

The latest release, current development versions, documentation, mailing lists, and FAQs are all available at the official site: `http://gcc.gnu.org`.

Our modifications lie solely within the semantic analyzer. In fact, the modifications to the existing codebase are less than five lines of code, most of them being declarations and comments for our own functions within standard headers. These few lines of code simply call into our own new routines, which do all the work (described in detail below).

GCC has several phases of compilation, and has multiple intermediate representations along the way:

1. The input source code is *preprocessed*, resulting in preprocessed source code.
2. The preprocessed source code is *parsed*, resulting in `tree`s that represent the code.

3. The `tree`s are semantically analyzed for correctness, changed as necessary for language semantics.
4. The `tree`s run through some optimization passes.
5. The `tree`s are converted to RTL, GCC's Register Transfer Language.
6. The RTL is optimized
7. The RTL is converted to machine code.

RTL is a fairly simple intermediate form that normalizes operations to the form ¡operation, parameter, parameter, destination¿, looking only slightly higher in abstraction than machine code. However, we did not go anywhere near it in our implementation, so there's no need to delve into it.

GCC's `tree` is a large and complex data structure. Firstly, it's not a typical data structure, the `tree` is actually a pointer to `struct tree_common`, which contains a `type`, `chain`, `code` and many flags. However, a variable of type `tree` rarely (if ever) points to a `tree_common` directly. Instead, other structure types are defined with a `tree_common` as their first member. The member `code` indicates which one of these types is actually being referenced.

`tree_common` is shown in Figure 4.1.

Such types include `tree_identifier` and `tree_vector`. Many of these types exist, describing the various language structures of GCC's front ends. For each type, macros act as type–safe accessors to internal data. For example, Figure 4.2 shows a complex constant tree type, `tree_complex`.

`tree_complex` has three data members. The first, `rtl`, is simply RTL information for the constant. The latter two are the real and imaginary parts of the constant. The two `#define`s above it are macros that first check the `tree_code` of the `common` segment, and then return the appropriate structure member.

The macro `COMPLEX_CST_CHECK` is automatically generated by the build system from the input file `tree.def`, which defines all the `tree_code` values, along with relevant metadata.

4.3 Template Instantiation

Templates are specified with some specific types omitted. Instead, a placeholder is used, which maps to a parameter for the template. When the template is to be used, all the parameters are filled with types. To build a complete definition for the type, the compiler makes a duplicate of the template definition, with the placeholder tokens replaced by their parameters, which is then fully compiled to machine code.

```
struct tree_common GTY(())
{
  tree chain;
  tree type;

  ENUM_BITFIELD(tree_code) code : 8;

  unsigned side_effects_flag : 1;
  unsigned constant_flag : 1;
  unsigned addressable_flag : 1;
  unsigned volatile_flag : 1;
  unsigned readonly_flag : 1;
  unsigned unsigned_flag : 1;
  unsigned asm_written_flag: 1;
  unsigned unused_0 : 1;

  unsigned used_flag : 1;
  unsigned nothrow_flag : 1;
  unsigned static_flag : 1;
  unsigned public_flag : 1;
  unsigned private_flag : 1;
  unsigned protected_flag : 1;
  unsigned bounded_flag : 1;
  unsigned deprecated_flag : 1;

  unsigned lang_flag_0 : 1;
  unsigned lang_flag_1 : 1;
  unsigned lang_flag_2 : 1;
  unsigned lang_flag_3 : 1;
  unsigned lang_flag_4 : 1;
  unsigned lang_flag_5 : 1;
  unsigned lang_flag_6 : 1;
  unsigned unused_1 : 1;
};
```

Figure 4.1: GCC's `tree_common` Structure

For example, for a generic type `Foo` as defined in Figure 4.3, `T` is a placeholder for a type. The declaration for `value` specifies a member variable.

Only when the parameters to a template are filled in is a full type specified. This specification process is called *instantiation*. In Figure 4.3, the variable `f` is a `class` of type `Foo<int>`

```
/* In a COMPLEX_CST node.  */
#define TREE_REALPART(NODE) \
          (COMPLEX_CST_CHECK (NODE)->complex.real)
#define TREE_IMAGPART(NODE) \
          (COMPLEX_CST_CHECK (NODE)->complex.imag)

struct tree_complex GTY(())
{
  struct tree_common common;
  rtx rtl; /* acts as link to register transfer
              language (rtl) info */
  tree real;
  tree imag;
};
```

Figure 4.2: GCC's `tree_complex` Structure

```
// the template class Foo
template<class T> class Foo {
    T value;
};

// an instantiation of Foo
Foo<int>  f;
```

Figure 4.3: Example Template

with a single member variable, an `int` called `value`. In C++, templates are the only class of types which have angle brackets (`<>`) in their names.

Upon instantiation, a compiler must convert the template definition into a normal type. Inside GCC, the instantiation process looks very similar to a search and replace (`s/T/int/g`) within the body of the template.

4.4 Modifications

Our modifications to GCC come in three parts: *detection*, *type analysis*, and *annotation*. When an attempt to use one of the compiler–generated classes is detected, new variables, enumerations, and type definitions are added to the template's instantiated definition. The values for the additions come from an analysis of the types given as template parameters.

```
#include <ctti>
struct Foo { /* ... */}

bool b = std::type_description<Foo>::is_pod; // ****
```

Figure 4.4: Example Use of an Introspective Class

For example, given the line marked with asterisks in Figure 4.4, the modifications execute the following steps:

1. They *detect* the attempt to use an introspective class template, `type_description`.
2. They *analyze* the parameter to the template, `Foo`.
3. They *annotate* `type_description` with an enumerated type, containing a value `is_pod`. We know what value to set `is_pod` from the type analysis phase.

4.4.1 Detection

As all of Introspective C++'s functionality comes from "special" templates, we simply catch all attempts to instantiate a template and, if it's one of ours, put in our own definitions instead. The rest of the compiler acts as if the template were simply specialized for every parameter type given to it.

GCC calls `instantiate_class_template` to instantiate every class template, so we simply added in a call to the primary driver routine `ctti_maybe_annotate`.

First, `ctti_maybe_annotate` would call `should_annotate`, which determines if `g++` was instantiating an introspective template. `should_annotate` not only determines if an introspective template's being instantiated, but which one (`type_description`, `member`, etc.). From there, `ctti_maybe_annotate` calls an appropriate annotation function for the introspective template. The annotation functions are all listed in Table 4.1.

In the example above, it would call `annotate_type_desc`.

Each of the functions listed in Table 4.1 analyze the template parameters given and then annotate the introspective class with what it found. All introspective classes are in `namespace std`.

The work for type analysis and annotation has been factored into a couple of utility routines, covered later. The annotation procedures amount to modifying the parsed, representation of the introspective types. The analysis is little more than querying the same representations for the required information.

Introspective Type	Annotation Function
`type_description`	`annotate_type_desc`
`type_description::base_class`	`annotate_baseclass`
`type_description::param`	`annotate_param`
`type_description::template_param`	`annotate_templparam`
`type_description::member`	`annotate_member`
`type_description::method`	`annotate_method`

Table 4.1: Annotation Functions

It's important to note that these additions depend on all the introspective class templates being declared properly, as they are in the header `ctti` (short for Compile–Time Type Introspection). When you're using the introspective types, remember to `#include <ctti>`.

4.4.2 Type Analysis

The type analysis code simply traverses the parameter type's `tree` for the desired information. They are listed in Table 4.2. The code consists of mostly query functions that return simple integer values.

Analysis Function	Purpose
`q_class(t)`	Is t a class type?
`q_union(t)`	Is t a union type?
`q_enum(t)`	Is t an enumerated type?
`q_function(t)`	Is t a function type?
`q_array(t)`	Is t an array type?
`q_complete(t)`	Is t completely defined?
`q_contained(t)`	Is t an inner type?
`q_align(t)`	What's t's alignment?
`q_bases(t)`	How many base classes does t have?
`q_template(t)`	Is t an instantiated template?
`q_templ_params(t)`	How many template parameters?
`q_members(t)`	How many members in t?
`q_func_params(t)`	How many function parameters?
`q_arrlen(t)`	How many elements in array t?

Table 4.2: Analysis Functions

For the most part, these functions just return values already maintained by the `tree t`. The last four functions have to do a bit of counting, but are still pretty short. Figure 4.5 shows the implementation for `q_func_params`.

```
static int
q_func_params (tree t) {
    if (TREE_CODE(t) == FUNCTION_TYPE) {
        int cnt=0;
        tree cur = TYPE_ARG_TYPES(t);
        while (cur && TREE_CODE(cur) != VOID_TYPE) {
            cnt++;
            cur = TREE_CHAIN(cur);
        }
        return cnt;
    } else {
        return 0;
    }
}
```

Figure 4.5: `q_func_params` from `ctti.c`

4.4.3 Annotation

Once the analysis has been complete, the definition for `type_description<T>` has to be filled in for `T`. To do that, we have to modify the `tree` that describes it. Table 4.1 lists the functions that do the annotation for each introspective type, which call the functions in Table 4.3 that do the actual tree manipulation.

Annotation Utility Function	Purpose
`append_decl`	Adds an initialized variable declaration.
`append_string`	Adds a static, constant string.
`append_typedef`	Adds a typedef.
`append_enums`	Adds an enumerated type definition.

Table 4.3: Tree Manipulation Functions

Chapter 5

Case Studies

5.1 Container Optimization

Having knowledge of a type's properties can help make code faster. To illustrate, we'll present a simple, grow–only dynamic array. This array will only allow the addition of items. However, it will need to grow its internal buffer as the additions come in.

To properly grow a buffer of objects, one has to make copies of the items in the old buffer into the new one, and then delete the old buffer. However, as this array is for arbitrary objects (it's a template), it will have to call the copy constructor for each object in the buffer during the copy operation. Then, it'll have to call the destructor for every object in the old buffer. For objects containing references to additional resources, like file handles, semaphores, or blocks of memory, this is all very necessary. However, for a structure containing an integer and a character, this is quite wasteful; the container could simply copy the buffer as a set of bytes and completely ignore the old buffer's objects before deletion.

A type which looks and acts like a C `struct` is called a POD type. PODs have no custom constructor, destructor, `operator=`, or virtual methods defined. All of their members must be POD as well. For these types, we can avoid calling the constructor and destructor altogether; we just need to worry about getting the bytes in memory correct.

Introspective C++ indicates if a type is POD with the enumeration `is_pod`. Using this enumeration, a container can decide whether or not to call the constructor or destructor at all. Other compilers, like [21], already use such an optimization for their own STL implementations.

5.1.1 Multiple Policies

In Figure 5.1, we have the two possible alternatives. The first, `vector_ops<T,N>`, treats the type `T` as if it was POD, bypassing the copy constructor and directly using `memcpy` to copy the bytes. `memcpy` is a standard library function for memory copies, often extremely optimized by the compiler vendor.

The second alternative, `vector_ops<T,0>` calls the copy constructor and destructor for every item. It's absolutely necessary when type `T` has its own resources to manage.

```
template<class T, int N>
struct vector_ops {
    void copy (T* dest, T* src_begin, T* src_end) {
        ::memcpy (dest, src_begin, src_end - src_begin );
    }

    void destroy (T* src_begin, T* src_end) {}
};

template<class T>
struct vector_ops<T,0> {
    void copy (T* dest, T* src_begin, T* src_end) {
        while (src_begin != src_end) {
            new (dest) T(*src_begin);
            src_begin++;
            dest++;
        }
    }

    void destroy (T* src_begin, T* src_end) {
        while (src_begin != src_end) {
            src_begin->~T();
            src_begin++;
        }
    }
};
```

Figure 5.1: Two Policies for Copying and Destroying Contents

5.1.2 Discrimination

To decide which policy to use for a given type, the container `vector<T>` will derive from the appropriate one, using `is_pod` as the discriminator. The simple diagram in Figure 5.2 illustrates the choice.

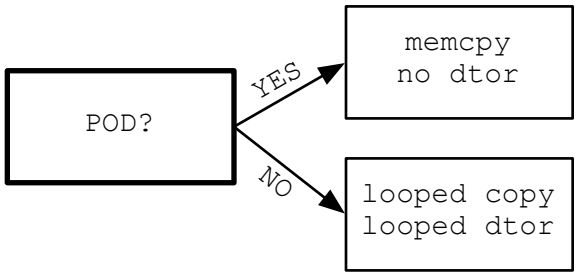


Figure 5.2: Overview of Policy Selection

Figure 5.3 shows `vector<T>`, with the operation `push_back`, which adds another element to the dynamic array.

Only when the dynamic array has to be resized does the policy matter. In that case, `copy()` has to copy the contents of the old buffer to the new one, and `destroy()` has to destroy the old buffer's objects.

5.1.3 Results

A simple test program is provided in the Appendix. It times calling `push_back` on a POD and a non-POD type for a given number of times. The iteration count is a command–line parameter. The left half of Figure 5.4 shows the definitions of both `yes_pod` and `not_pod`. Note that they have the same members, and thus have the same size. The only difference is in the new copy constructor, which forces the container to use the non–POD policies. The actual work required for both is the same: the integer and the character just need a bit–for–bit copy.

On my computer, a 1.25GHz PowerBook G4, the differences were significant, as in the right half of Figure 5.4. The program was compiled with maximum optimizations (`-O6`) on a modified (to add introspection) version of g++ 3.3.4. The program was also run several times before these, in each configuration, to get the VM to act smoothly.

```
template<class T>
class vector
  : private vector_ops<
       T,
       std::type_description<T>::is_pod
    > {
    T   *storage;
    int capacity, size;
public:
    vector () : storage(0), capacity (0), size (0) {}
    ~vector () { delete storage; }
    void push_back (const T& value) {
        if (size == capacity) {
            T * new_store;
            capacity = (capacity?
                  (capacity + capacity/2) : 16 );
            new_store = (T*) malloc (sizeof (T) * capacity);
            copy (new_store, storage, storage+size);
            destroy (storage, storage+size);
            delete storage;
            storage = new_store;
        }
        new (storage + size++) T(value);
    }
};
```

Figure 5.3: `vector`, Subclassing the Appropriate Policy

5.2 Runtime Access

The data provided in Introspective C++ can be preserved for runtime use. This example preserves only a subset of the data provided in Introspective C++, partially for the sake of readability, and partially to avoid certain complexities[1]. The relevant techniques, however, are present here, and can be used to extend this example to a full run–time support system. Examples of other runtime support systems are in [70, 58, 66].

[1]Mostly in the form of C++ bugs in `g++`

```
struct yes_pod {
    int val1;
    char val2;
};

struct not_pod {
    int val1;
    char val2;
    not_pod () :
      val1(0),
      val2(0) {}
};
```

```
-- Five Million Objects --
$ ./a.out 5000000
yes_pod: 0.593089 seconds
not_pod: 1.067204 seconds
$ ./a.out 5000000
yes_pod: 0.542528 seconds
not_pod: 1.163791 seconds
$ ./a.out 5000000
yes_pod: 0.577424 seconds
not_pod: 1.072614 seconds
$ ./a.out 5000000
yes_pod: 0.588105 seconds
not_pod: 1.149546 seconds

-- Fifty Million Objects --
$ ./a.out 50000000
yes_pod: 3.121570 seconds
not_pod: 8.361282 seconds
$ ./a.out 50000000
yes_pod: 3.071006 seconds
not_pod: 8.413466 seconds
$ ./a.out 50000000
yes_pod: 3.128506 seconds
not_pod: 8.290320 seconds
$ ./a.out 50000000
yes_pod: 3.089732 seconds
not_pod: 8.341073 seconds
```

Figure 5.4: The datatypes and their serialization times

5.2.1 Preserved Data

Figure 5.5 shows the data preserved here. It's essentially a linked–list of names and types for members, a linked–list of names for methods, and a top–level class description containing head pointers to the member and method lists. It doesn't provide pointers to methods or offsets to members, or cover overloaded methods. However, building the lists and traversing the Introspective C++ types are the important points of the example, and they're present.

```
struct method_info;
struct member_info;

struct class_info {
    const char * name;
    int nr_methods;
    int nr_members;
    method_info  *methods;
    member_info  *members;
};

struct method_info {
    const char * name;
    method_info *next;
};

struct member_info {
    const char * name;
    const char * type;
    member_info *next;
};
```

Figure 5.5: Runtime Introspective Mapping

5.2.2 Member and Method Preservation

As before, we'll use template recursion for traversing the list of members and methods. This also includes a specialization for the last entry to end the recursion. Also, we want to make the generated code as efficient as possible, that means as much work has to be pushed to the optimizer as possible. The key challenges are keeping all the memory allocations static and the initializations simple; preferably within the initialized data segment.

To create all the necessary instances of `method_info` and `member_info`, we use a recursive subclass of each. Each of their own instantiations begin from a subclass of `class_info`. The structure is shown in Figure 5.6.

Figure 5.7 shows `method_gen`, a subclass of `method_info` that defines a default constructor, filling in members using the introspective API. The first definition is the general case, the latter ends the recursion and sets the `next` pointer to zero.

In the first definition, the next element in the recursion is a member, letting all of the elements exist as a single allocation. How that allocation occurs, be it on the stack, heap, or static data segment, is defined by the client. The latter definition has no additional members,

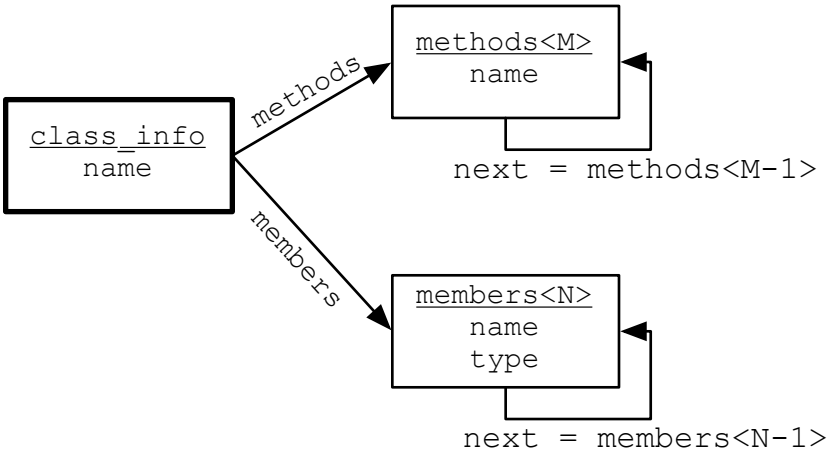


Figure 5.6: Overview of the Preservation Process

```
template<class T, int N> struct method_gen : public method_info {
    typedef typename std::type_description<T> typedesc;
    typedef typename typedesc::method<N-1> method_desc;
    method_gen<T,N-1> next_meth;
    method_gen () {
        name = method_desc::name;
        next  = & next_meth;
    }
};

template<class T> struct method_gen<T,1> : public method_info {
    typedef typename std::type_description<T> typedesc;
    typedef typename typedesc::method<0> method_desc;
    method_gen () {
        name = method_desc::name;
        next = 0;
    }
};
```

Figure 5.7: `method_gen` to Initialize `method_info`

and sets `next=0`.

The definitions for members are similarly in Figure 5.8.

```
template<class T, int N> struct member_gen : public member_info {
    typedef typename std::type_description<T> typedesc;
    typedef typename typedesc::member<N-1> member_desc;
    member_gen<T,N-1> next_mem;
    member_gen () {
        name  =  member_desc::name;
        type  =  std::type_description<
                   typename member_desc::type
                  >::name;
        next = & next_mem;
    }
};

template<class T> struct member_gen<T,1> : public member_info {
    typedef typename std::type_description<T> typedesc;
    typedef typename typedesc::member<0> member_desc;
    member_gen () {
        name  =  member_desc::name;
        type  =  std::type_description<
                   typename member_desc::type
                  >::name;
        next = 0;
    }
};
```

Figure 5.8: `member_gen` to Initialize `member_info`

5.2.3 Class Data Preservation

Finally, we cover `introspective_generator`. It houses the non–recursive `class_init`, which initializes its base class `class_info`. Finally, it declares an instance of `class_init` called `generated_class_info`. Through `class_init`'s `gen_members` and `gen_methods` methods, the method and member lists are generated automatically when `introspective_generator` is instantiated.

5.2.4 Usage

To show it all in action, we'll look at an example source file that defines a data type and then prints out what it gets from the preserved introspective information. See Figure 5.10.

We statically generate the introspective information for `introspect_me` in the declaration of

```
template<class T> struct introspective_generator {
    typedef typename std::type_description<T> typedesc;
    struct class_init : public class_info {
        method_gen<T, typedesc::num_methods> gen_methods;
        member_gen<T, typedesc::num_members> gen_members;
        class_init () {
            name =typedesc::name;
            nr_methods =typedesc::num_methods;
            nr_members =typedesc::num_members;
            methods = & gen_methods;
            members = & gen_members;
        }
    } generated_class_info;
};
```

Figure 5.9: `introspective_generator`

`gen` in `main`. That's fully generated before `main` runs, and can then be directly used in the declaration of `ci`. The resulting data structure created through this process is illustrated in Figure 5.11.

The remainder of the code is simple to understand, and so we just provide output in Figure 5.12.

5.3 Serialization

Serialization [38, 48] is useful and isn't already built into C++. As such, it makes a good starting example. The strategy is simple: we define two serializers, one for input types that can directly be written to disk, and another that decomposes its input. Hopefully the latter will ultimately decompose completely into types that can be written by the former. The decomposing serializer we'll call `Serializer`, and the direct serializer will be called `PrimitiveSerializer`.

To determine which serializer to use for a given type, we define a type traits class, `SerializeInfo`, which has a typedef called `serializer`. This typedef will either be to `Serializer` or `PrimitiveSerializer`, depending on `type_description<T>::is_pod`. Recall that a type is considered POD if it's either a primitive type or an aggregate of PODs with no virtual methods and default definitions of the default and copy constructors, `operator=`, and the destructor. Figure 5.13 illustrates the process.

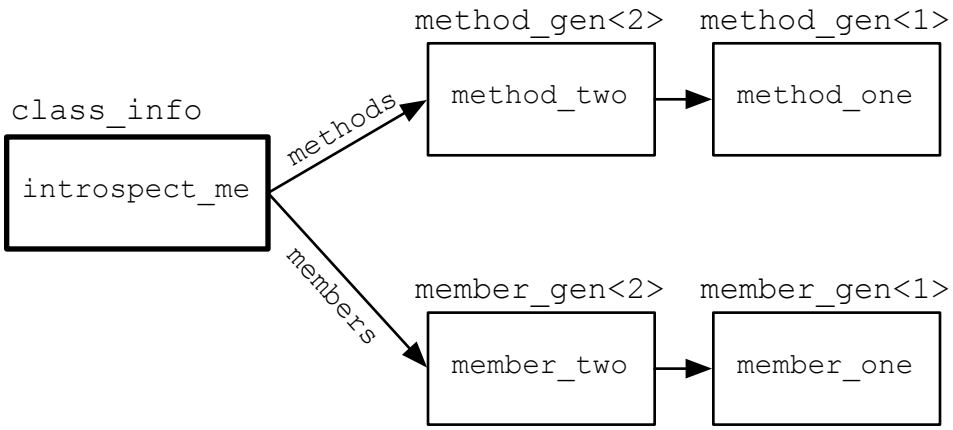
As `SerializeInfo` is a type traits class, it can be redefined for any other type, so that other,

```
#include "rttd.h"
#include <stdio.h>

struct introspect_me {
    int  member_one;
    int  member_two;
    void method_one () {}
    void method_two () {}
};

void print_member (member_info *m) {
    while (m)
        printf ("\tMember: \"%s\": %s\n",
                 m->name,
                 m->type), m=m->next;
}

void print_method (method_info *m) {
    while (m)
        printf ("\tMethod: \"%s\"\n",
                m->name), m=m->next;
}

int main (int args, char ** argv) {
    static introspective_generator<introspect_me> gen;
    class_info *ci = & gen.generated_class_info;

    printf ("Class \"%s\"\n", ci->name);
    print_member (ci->members);
    print_method (ci->methods);
    return 0;
}
```

Figure 5.10: Using the Preserved Data

custom serializers could be used instead. These serializers need not even be templates. We use this capability to prevent automatic serialization of pointer types, as they'd be senseless once the serialized form has left the original program's address space.

To support the serialization of a specific pointer type, `SerializeInfo` needs to be specialized for it, with a new serializer defined. We provide an example of specialization with `const char*`, as seen below.



Note: compiler-generated methods aren't shown here.

Figure 5.11: Resulting Data Structures

```
Class "introspect_me"
        Member: "member_two": int
        Member: "member_one": int
        Method: "method_two"
        Method: "method_one"
        Method: ""
        Method: ""
        Method: "operator="
        Method: "introspect_me"
```

Figure 5.12: Output of Preserved Data use

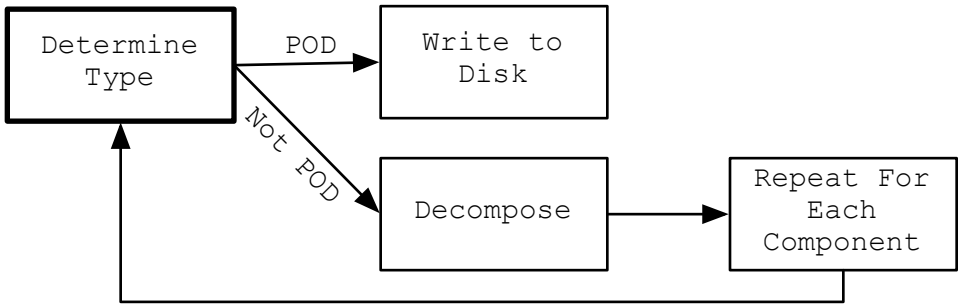


Figure 5.13: Overview of the Serialization Process

The destination, `class bytestream`, is a very simple class that uses the stream insertion (`<<`) and extraction (`>>`) operators like the standard `iostream` mechanism, but uses a binary representation on disk instead of a textual one. Its source is provided as an appendix.

5.3.1 Primitive Types

The serializer for primitive types is very simple, as shown in Figure 5.14. It's been overridden for `const char *`, as shown in Figure 5.15.

```
// works for any POD type.
template<class T> struct PrimitiveSerializer {
    static void put (const T& value, bytestream& dest) {
        printf ("serializing type %s\n",
            std::type_description<T>::name);
        dest << value;
    }

    static T    get (bytestream& src) {
        T result;
        src >> result;
        return result;
    }
};
```

Figure 5.14: `PrimitiveSerializer`

`PrimitiveSerializer` is only intended to work for simple POD types, whose binary representations in memory can directly be encoded to the bytestream, and visa–versa.

5.3.2 Composite Types

For types which aren't POD, but can still be automatically decomposed, we have `Serializer`. `Serializer` uses introspection to get all the members of its parameter type, and then serializes them individually.

To work for a variable number of members, `Serializer` uses a recursive helper type called `SerializeWorker`. `SerializeWorker` invokes the individual serializations of each member. It takes in two parameters, the container type `T` and the index of the member to work on `N`. Note that it actually works on member `N-1`, as the members are zero–indexed, while we use zero as a terminator for the recursion (shown later). See Figure 5.16.

Don't worry about `SerializeInfo` yet, we'll cover that in the next section. However, we have to stop the recursion once we run out of members, and for that we have the specialization of `SerializeWorker`for `N=0`, in Figure 5.17.

With the two definitions above, we can understand the definition for `Serializer`. It's pretty simple, as seen in Figure 5.18.

```
template<> struct PrimitiveSerializer<const char*> {
    static void put (const char *value, bytestream& dest) {
        unsigned int len;
        printf ("putting \"\%s\"\n", value);
        if (!value || !*value) {
            len = 0;
            dest << len;
        } else {
            len = strlen (value);
            dest << len;
            while (len--)
                dest << *value++;
        }
    }
    static const char * get (bytestream& src) {
        unsigned int len;
        src >> len;
        if (len) {
            char *p, *dest = (char*) malloc (len+1);
            p = dest;
            len++; // terminating zero!
            while (len--)
                src >> *p++;
            return dest;
        } else {
            return 0;
        }
    }
};
```

Figure 5.15: `PrimitiveSerializer` for C Strings

All `Serializer` does is start the recursion, using introspection to get the number of members within the type to initialize `N`. `SerializeWorker` will serialize each member and recurse downwards to `N=0`.

5.3.3 Selection

Now that we have mechanisms for primitive and composite types, we need a method of deciding which one to use at any given instant. For that, a type traits class is developed,

```
template<class T, int N> struct SerializeWorker {
    static void put (const T& value, bytestream& dest) {
        typedef typename std::type_description<T>
                 ::template member<N-1>
                  member_info;
        SerializeInfo<
          typename member_info::type
        >::result::put (
                &(value.*member_info::ptr),dest
            );
        SerializeWorker<T,N-1>::put (value, dest);
    }
    static void get (T& dest, bytestream& src) {
        typedef typename std::type_description<T>
                 ::template member<N-1>
                  member_info;
        src.*member_info::ptr = SerializeInfo<
                 typename member_info::type
              >::result::get (dest);
        SerializeWorker<T,N-1>::get (dest, src);
    }
};‘
```

Figure 5.16: `SerializeWorker` – Normal Definition

```
template<class T> struct SerializeWorker<T,0> {
    static void put (const T& value, bytestream& dest) {}
    static void get (T& dest, bytestream& src) {}
};
```

Figure 5.17: `SerializeWorker` – Recursion Terminator

`SerializeInfo`. This traits class has a default implementation (overridable through template specialization), that uses the `is_pod` property of the introspection mechanism to decide. It's listed in Figure 5.19.

5.3.4 Execution

To demonstrate this system in action, we present a simple data structure, consisting of a single integer and a C string (`char*`). We give it a specialized constructor so that it isn't a

```
template<class T> struct Serializer {
    static void put (const T& value, bytestream& dest) {
        SerializeWorker<T,
            std::type_description<T>::num_members
        >::put (
           value, dest
           );
    }

    static T  get (bytestream& src) {
        T result;
        SerializeWorker<T,
           std::type_description<T>::num_members
        >::get (
           result, src
           );
        return result;
    }
};
```

Figure 5.18: `Serializer`

POD type. The type is shown in Figure 5.20, along with some code to invoke serialization upon it. The output of that serialization process is shown in Figure 5.22.

To understand the output, Figure 5.21 illustrates the templates in use. As the decomposition occurs from last member to first, the C string `sval` is serialized first. This invokes the specialized `PrimitiveSerializer` for `const char*`[2].

The specialized serializer for `char*` first writes the length of the string (a four byte integer, [`00 00 00 03`]), and then the string itself (three bytes of ASCII-encoded text [`66 6f 6f`]). Finally, the member `ival` is written using the default serializer (the last four bytes [`de ad be ef`]).

[2]As seen in the full source code for `serializer.h`, `SerializeInfo` for `char*` has been specialized to return the same value as `const char*`.

```
//  Will switch between Serializer and PrimitiveSerializer,
// used by SerializeInfo.
template<class T, int N> struct SerializerPODSwitch {
    typedef PrimitiveSerializer<T> result;
};

template<class T> struct SerializerPODSwitch <T,0> {
    typedef Serializer<T> result;
};

// Determines the proper Serializer type. Override as needed.
template<class T>
struct SerializeInfo {
    typedef typename SerializerPODSwitch<
                        T, std::type_description<T>::is_pod
                      >::result
                 serializer;
};
```

Figure 5.19: `SerializeInfo` and it's Helper `SerializerPODSwitch`

```
struct data {
    int ival;
    char *sval;
    data () { ival = 1; sval = "data"; }
};

int main (int args, char **argv) {
    data d;
    d.ival = 0xdeadbeef;
    d.sval = "foo";
    bytestream bs ("foo.out");
    SerializeInfo<data>::serializer::put (d,bs);
    puts ("writing to foo.out");
    return 0;
}
```

Figure 5.20: Example Type to be Serialized

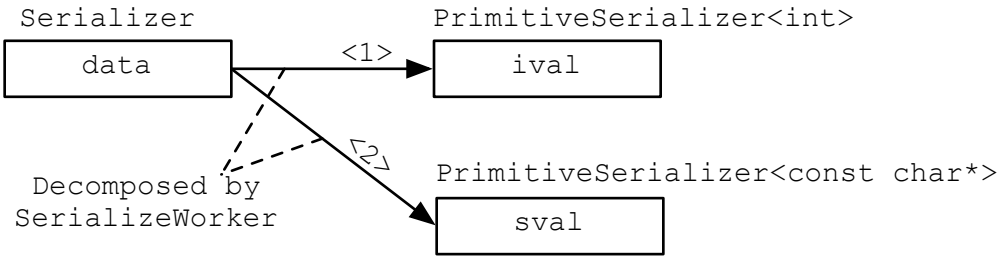


Figure 5.21: Serialization Process of `struct data`

```
0000    0003    666f    6fde    adbe    ef
```

Figure 5.22: Hex dump of `foo.out`

Chapter 6

Related Work

Introspective C++ is not the first attempt to add introspection to C++. Below, we consider three prior attempts: a "Non–Intrusive Object Introspection" mechanism in C++ [12], XVF: The eXtensible Visitation Framework [74], and the Iguana system [33].

6.1 Non-Intrusive Object Introspection in C++

Chuang et al [12] describe a run–time introspection mechanism for C++ in a paper entitled *Non-Intrusive Object Introspection in C++*; we'll use the abbreviation NIOI to identify it. This mechanism involves a preprocessor that parses type declarations and generates metaclasses for them. These metaclasses are loaded with the program and are available through runtime libraries.

The metaclasses provide structural introspection, with the ability to call methods whose names and recipients are determined at run–time. With no "root object" like Java [45], the mechanisms use a base type of `void*`, in essence bypassing the C++ type system.

6.1.1 Comparison

NIOI uses a significantly different mechanism for introspection, and comparison with Introspective C++ is both warranted and revealing.

Overall, NIOI takes a very straightforward "one size fits all" approach — provide functionality that many customers will want in a simple, fairly automatic way. Introspective C++ takes a different approach, closer to "we don't know what you want, so we'll just give you the tools to build it yourself." The two approaches fit very different requirements. Actually, the two don't even preclude one another — both could be used within the same system.

Advantages

NIOI provides a quick, simple mechanism for many traditional C++ applications to quickly add run–time structural introspection. Consideration is even given to complex scenarios such as shared library integration [12]. The mechanism provides a turnkey mechanism for quickly enabling many often–used constructs of introspection: getting a metaclass from a name, invoking a method that was only identified at runtime, and the runtime discovery and access of object members.

The system provides run–time checks to compensate for the lost type information. As structural introspection is the discovery of otherwise–unknown type information, such checks will inevitably be necessary.

Consequences

The consequences of NIOI are some of the very ones we aimed to avoid at the very beginning of the design of Introspective C++. This does not discourage use of NIOI in any major way: the intended usage and requirements to be satisfied for each are quite different.

First, the entire metaclass is generated for every nontemplate type in the system. For many systems [45,14], this is perfectly acceptable. However, some systems may have more stringent memory and storage requirements, or simply too small a need for introspection to justify the additional overhead. In those cases, NIOI would be inappropriate.

Second, a second tool is used to generate the metaclass information. While certainly better than hand–written descriptions, an external tool requires a second parser. That second parser has the potential to interpret the code differently from the compiler's parser, due to bugs in either the tool or compiler, or due to accidental use of ambiguous or nonstandard constructs within the language. C++ is a complex language and interpreting the full syntax is nontrivial.

The second tool also causes NIOI to have less information than always necessary. Chuang et al explicitly mentions the requirement for explicit specification of which template instantiations the tool should generate metaclasses for, in the form of "hints" [12]. This requirement adds one side benefit — programmer control over which template instantiations get metaclasses generated for them. In the factorial example in Section 3.1.3, this allows the programmer to specify that a metaclass is only generated for `factorial<3>`, but not for `<2>`, `<1>`, or `<0>`. Unfortunately, it mandates that the programmer knows and maintains a list of all the necessary template instantiations that the introspective mechanism will use. For simple cases, this isn't a problem. As the codebase adds more metaprogramming [2] constructs, this could get increasingly complex.

Finally, NIOI is a solely run–time system. It does not present an interface for manipulation within the type and template systems. As such, potential optimizations (such as determining

statically that a type has to have a given member or method) aren't available. As shown in the completely static serialization mechanism in Section 5.3, it can be possible to completely resolve introspective questions at compile time using templates.

6.2 XVF: eXtensible Visitation Framework

Stephen's XVF: eXtensible Visitation Framework provides a uniquely simple and flexible mechanism for structural introspection. A set of macros are inserted into class definitions that expand out to a run–time meta–object protocol. Specifically, the definitions expand out to methods and definitions that enable the use of the Visitor pattern [30] to traverse the definition of the type.

The system requires no external tools or language extensions. It provides complete programmer control over which types have metadata generated for them. It defines the metadata within the original type, easing management and removing any additional complexity that could otherwise occur when interacting with dynamic linkers [12].

6.2.1 Comparison

XVF is a simple, straightforward, portable, understandable framework using only standard C++. Comparison against Introspective C++ provides a suitable litmus test for our work.

Advantages

XVF doesn't require any external tools or modifications to the compiler. As such, the C++ programmer already has the tool, build system, and vendor support in place. The code is inherently portable due to its compliance to ANSI C++. The Visitor pattern is well–known, well suited for structural introspection, and understandable.

Consequences

Introspective C++ does have its advantages. XVF requires that the programmer call macros in and around each introspected type's definition. These macros depend not only on the declaration of the type, but the internal definition: members and methods. As such, the type definition given to XVF through these macros can come out of synchronization with the one used. Such problems can be troublesome and time–consuming to detect and repair.

Like Chuang et al [12], XVF provides only a run–time mechanism. It shares the same consequences: no opportunity for compile–time resolution and optimization and limited

usability for template components.

Finally, it's unclear how or if XVF can be used to introspect instantiations of a template. It uses macros that must be called outside of the type definition (specifically, `XVF_CLASS_BEGIN`, `XVF_CLASS_END`, and `XVF_CLASS_INSTANCE`), which would not be part of the definition of the instantiated template.

6.3 Iguana

The Iguana [33] syste shares many similar goals with Introspective C++. It provides a MOP [47] for C++, but without the traditional overhead of preserving all information for all data types.

Instead, Iguana provides a metatype system using a precompiler. The metatype system allows the definition and use of MOPs. These MOPs can even be defined for new forms of method dispatch[1]. Each MOP is attached to an object via a new keyword `reify`, followed by a type decaration and an optional variable name. A default name is provided if unspecified.

That variable is accessible at run–time, as an instance of the MOP for that object. Multiple MOPs may be attached to the same object. MOPs are defined in a syntax similar to C++ classes without methods. Inheritance, local variables, prerequisite MOPs are allowed.

Gowing and Cahill [33] have a similar goal as ours: "to avoid falling into the 'reify everything' trap." As such, the default MOP is an empty MOP, which reifies nothing.

6.3.1 Comparison

Iguana provides most of the functionality of Introspective C++ without the complexity of template syntax or, in many cases, the need to write one's own MOP at all. Iguana provides 23 different reification categories, each a MOP in itself. That library of MOPs is sufficient for many problems as is.

Advantages

Iguana clearly can make a programmer's job easier simply by reuse of the existing libraries. Specifying the MOP to use is through a new keyword, `reify`, which results in another public instance variable on the object. The preprocessor provides a new but fairly natural language for C++ programmers to specify new MOPs.

[1] Although it should be noted that the new dispatch requires some additional syntax: `obj->meta->reception->receive(...)`.

Consequences

Iguana works using a preprocessor. As such, it is a separate tool that requires its own compilation and build–system overhead. It's unclear how the preprocessor would interact with other language features such as namespaces and templates. As of the writing [33], the parser did not parse all of the C++ base language, although work was already in progress to use the Brown University Cppp C++ front–end parser.

Without integration with the template mechanism, Iguana would not allow the C++ compiler to resolve introspection at compile–time. As Iguana doesn't provide a compile–time introspective mechanism, it doesn't allow templates to use introspection during their instantiation. So, there will be some components using Iguana that require run–time memory and execution overhead that could have been implemented in Introspective C++ without it.

Chapter 7

Conclusions

Introspective C++ extends C++ from simple type identification to structural introspection. It allows static analysis of types to minimize run–time introspective overhead. It does not impose a "one size fits all" run–time mechanism, nor the associated overhead. Furthermore, it does not impose its mechanism upon any specific type or class of types.

With all the non–impositions, it allows the programmer to design their own run–time introspective mechanisms as needed. The programmer can choose which types provide structural introspection, and the degree and format of that introspective data. Furthermore, multiple mechanisms can coexist within the same system, allowing tight optimization while still being able to leverage all the introspective capability.

Introspective C++ leverages the inherent capabilities already within C++. It leverages the generics system to implement an efficient introspective mechanism that maximizes the potential for static resolution of introspective problems. It also enables maximal use of the compiler's optimizer to minimize introspective costs.

Unlike Java's introspection, which imposes such significant runtime overhead that a logarithmic graph is required [71], Introspective C++'s features impose no great runtime overhead. By default, it imposes none whatsoever. When [71] compared method dispatching through introspection and through a direct call in Java, it found a 10x speed hit for using introspection. In contrast, Introspective C++'s features a direct pointer to a method or member via the respective `ptr` members. The methods can be called through a single function pointer dereference, which is even faster than the double pointer dereference required for C++'s normal polymorphic mechanism.

Metaprogramming techniques are already used to implement optimization passes like loop unrolling [79], and similar passes could take advantage of information retrieved through the compile–time introspective mechanism. These techniques would then benefit from the compiler's optimization passes, which results in much compile–time resolution of introspective evaluations.

7.1 Limitations and Trade Offs

Introspective C++ provides only static introspection, through a static interface. While a dynamic interface can and has been built atop of Introspective C++ to provide non–template, run–time access to the data available, we are still limited to static introspection.

7.1.1 No Dynamic Introspection

To achieve any form of dynamic introspection, we need to build proxies of other classes. A proxy to an object has all of the same methods as the object itself, and receives messages for that object. The proxy will often pass these messages on to the object. With a proxy, a program can keep track of what messages are sent to an object, enabling dynamic introspection of its invocation history. If the object's connections to others are proxied as well, the program can keep track of the object's call history as well.

However, proxies aren't possible in Introspective C++. While C++ does have polymorphism, it still depends on the names of methods to be known at compile time. That mandates that any proxy component that can proxy more than one type be a template. But, even a template cannot specify an identifier that was not known at compile time. C++, and by extension, Introspective C++, doesn't provide a mechanism to construct an identifier through any compile–time mechanism. Proxies don't have to construct an identifier piece–by–piece, but they do have to replicate the interface of their object. Through Introspective C++, it's possible to specify an interface with the same number of public methods and members of another type. But, it's not possible to give them the names of those methods and members.

Even if it was possible to define a method or member who's name was determined through introspection, a similar problem occurs when attempting to duplicate a method parameter list. Introspective C++ provides a method's parameter list, but has no means for assembling them into a formal parameter list declaration of another method.

Even if we were able to both duplicate a method's name and formal parameter specification, we would often have to invoke the original method on the original object. Such an invocation would require that we be able to build a method call. A method call has two parts, a method name and a list of parameters separated by commas and wrapped in parenthesis.

As we're assuming that we can duplicate names, we can get the name of the method. However, there isn't a mechanism in either C++ or Introspective C++ that allows the assembly of a parameter list for a method call. There are ways around this last restriction, if the compiler has an extension for inline assembly, and if that extension could be used in a template. If these are true, it can be possible to devise a template that pushes the parameters onto the stack manually[1] and causes the call to the method's code. However, such a template would

[1] That is, through inline assembly language commands.

be processor and link–format specific.

7.1.2 A Compile–Time Mechanism

Introspective C++ uses a purely compile–time mechanism. It presents no run–time interface whatsoever. Clearly many applications need a run–time interface. The rationale has been mentioned in parts across this document, but it's best to summarize it now:

1. *A Run–time Interface Can be Constructed* — Using the template interface, it's straightforward to reify needed inforamtion for run–time access.

2. *Allows Programmer Selection of MetaObjects* — As Introspective C++ doesn't generate run–time metaobjects, it doesn't force an arbitrary selection of which types have metaobjects generated. As the generation of metaobjects isn't difficult, the programmer can generate them for the types they want.

   For C++, it's especially important that metaobjects don't get generated for every type. The language has a tradition and user base that frowns on execution or memory overhead. Equally important, metaprogramming can generate many templates, most of them are simply temporary values used for other metaprogramming. As such, the metaobjects for these intermediary template instantiations would be a complete waste. For sophisticated metaprogramming, often aimed at high degrees of optimization, generating these metaobjects would be prohibitively wasteful.

3. *Enables Early Resolution When Possible* — Thanks to the template mechanism, it's possible to define components as templates. Such components can be flexible in, and well decoupled from, their connections to other components, yet incur no runtime overhead for it through tight coupling in the executable. As a compile–time mechanism, Introspective C++ allows introspection to occur as the introspecting template components are being instantiated, allowing them to tightly bind to the results of the introspective process.

   Through this pairing of loose source binding and tight binary binding — having components defined in source code with little coupling to others, while having compiled binary code with tight coupling to them — we allow both easier maintenance and higher efficiency. By enabling those same template components to use introspection, they can resolve as many introspective operations at compile time as possible, minimizing run–time overhead for introspection.

4. *Enables Strong Type Checking* — As the type information isn't lost at compile time, the compiler can still validate the type consistency of the code. Run–time systems like the Non–Intrusive Object Introspection for C++ [12] provide their interfaces through untyped (such as `void*`) interfaces.

Unfortunately, it leaves a large problem: while they can build their own as they like, the programmer doesn't have a ready–made, run–time introspective mechanism available.

The case study in Section 5.2 illustrates how to implement one. However, it's just an example, not a full–blown system.

7.2 Future Work

The limitations and tradeoffs listed in the prior section guide our future work quite well. First and foremost, a set of libraries that implement different forms of run–time introspection are necessary. The Iguana [33] system has a set of different systems defined, and those definitions will serve as a good starting point for deciding what kinds of mechanisms to provide.

With those libraries in place, we can start looking at adapting the C++ syntax to enable the generation of proxy [30] classes. Enabling calculated identifiers and opening parameter list definitions and use for template metaprogramming would do just that, but it would entail changes to the C++ syntax. The hope is that we can use a syntactical construct which is currently illegal, preferably so blatantly illegal that no compiler currently accepts it by accident. With such a construct, we can be reasonably sure that we don't break any existing code. Hopefully, C++ programmers can adapt to the new construct quickly, perhaps inspired to learn it from its syntactic audacity.

Appendix A

Compiler Modifications Source

A.1 ctti

```
/** @file ctti
 *  This is a header for GCC's Introspective Extension.  You
 *  should #include this header when you want to use
 *  Compile-Time Type Introspection.
 */

#ifndef _GLIBCXX_CTTI
#define _GLIBCXX_CTTI 1

#pragma GCC system_header
namespace std {
    /* don't change the name or the namespace it exists in,
       g++ depends on these matching its internals */
    template<class T> class type_description {
    public:
      /* For convenience only -- g++ will put
         these in.
        enum { align, is_pod, array_length,
             is_class, is_union, is_enum,
             is_function, num_parameters,
             num_templ_parameters,
             num_members, num_methods };
        typedef xx inner_type;
        typedef xx return_type;
```

```
static const char * name; */

/* base class descriptions */
template<int N> struct base_class {
/*  typedef xx base_type;
    enum { is_virtual }; */
};

/* function parameter descriptions */
template<int M> struct param {
/*  typedef xx value_type; */
};

/* template parameter descriptions */
template<int O> struct template_param {
/*  typedef tp_* param_type; // only tp_int
                             // or tp_class
    enum { value_cst = VALUE_IF_PARAM_IS_AN_INT };
    typedef xx value_type; // if param is a type */
};

template<int P> struct member {
/*  typedef xx type;
    const static char *name;
    enum {is_static};
    type * ptr; // pointer-to-(sometimes static)
                // member */
};

template<int Q> struct method {
/*  enum { is_const, is_virtual, is_static,
           is_nothrow, num_overloads, is_ctor,
           is_dtor, is_operator };
    typedef xx type;  // a method type
                      //   (like a function type)
    static type *ptr; // pointer to method.

    const static char *name; */

    template<int R> struct overload {
    /*  enum { is_const, is_virtual, is_static,
               is_nothrow };
```

```
                typedef xx type;  // a method type
                                  //  (like a function type)
                static type *ptr; // pointer to method. */
            };
        };

        enum template_param_type_t {
            tp_class,
            tp_typename = tp_class,
            tp_int,
            tp_enum = tp_int,
        };
    };
}

#endif
```

A.2 ctti.c

```
/* Implements Compile-Time Type Information (CTTI).
   Copyright (C) 2003 Free Software Foundation, Inc.
   Written by Lally Singh (lallysingh@mac.com)

This file is part of GNU CC.

GNU CC is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public
License as published by the Free Software Foundation;
either version 2, or (at your option) any later version.

GNU CC is distributed in the hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the
implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.  See the GNU General Public License
for more details.

You should have received a copy of the GNU General
Public License along with GNU CC; see the file COPYING.
 If not, write to the Free Software Foundation, 59
```

```
Temple Place - Suite 330, Boston, MA 02111-1307, USA.
*/

/* Strategy
   --------

   Programs that want to use CTTI #include <ctti>.  That
 introduces the names of the CTTI types into the program.
 They're all templates.   When g++ tries to instantiate
 any of them, it calls ctti_maybe_annotate, which notices
 that it's a CTTI type, and fills in a definition. */


#include "config.h"
#include "system.h"
/* #include "coretypes.h" */
/* #include "tm.h" */
#include "obstack.h"
#include "tree.h"
#include "flags.h"
#include "cp-tree.h"
#include "tree-inline.h"
#include "decl.h"
#include "lex.h"
#include "output.h"
#include "except.h"
#include "toplev.h"
#include "rtl.h"
#include "timevar.h"
#include "tree.h"
#include "ggc.h"

/* only API exported from this file.  Driver for the rest */
tree        ctti_maybe_annotate( tree );

/* annotation drivers */
static int  should_annotate (tree);
static tree annotate_type_desc (tree);
static tree annotate_baseclass (tree);
static tree annotate_templparam (tree);
static tree annotate_param (tree);
static tree annotate_any_method (tree, tree, tree);
```

```
static tree annotate_method (tree);
static tree annotate_method_overload (tree);
static tree annotate_member (tree);

/* utility functions */
struct enum_val {
    const char * name;
    int value;
};

static tree get_integer( int );
static tree nth_base( tree, int );
static tree nth_method (tree, int, int *);
static int  is_member( tree );
static tree pull_name_id( tree );
static const char * pull_name( tree );
static int  pull_templ_int (tree, tree*, int*);
static tree append_enums( tree, struct enum_val *, int );
static tree append_typedef( tree, const char *, tree );
static tree append_string( tree, const char *,
                           const char *);
static void append_decl (tree, tree, tree, tree);

/* enum generation functions */
static int  q_class (tree);
static int  q_union (tree);
static int  q_enum (tree);
static int  q_function (tree);
static int  q_array (tree);
static int  q_complete (tree);
static int  q_contained (tree);
static int  q_align (tree);
static int  q_bases (tree);
static int  q_template (tree);
static int  q_templ_params (tree);
static int  q_members (tree);
static int  q_methods (tree);
static int  q_func_params (tree);
static int  q_arrlen (tree);

/* Method Enum Generators */
static int  q_trivial (tree);
```

```
static int  q_static (tree);
static int  q_const (tree);
static int  q_virtual (tree);
static int  q_operator (tree);
static int  q_nothrow (tree);

/*
 TYPE ANALYSIS SECTION
 ---------------------

 Almost all analysis of the parameter type
 occurs within these functions below.  They
 are categorized by the types they analyze.

 */

#pragma mark --Enum Generation Functions--

static int
q_class (tree tr)
{
    return TREE_CODE(tr) == RECORD_TYPE;
}

static int
q_union (tree t)
{
    return TREE_CODE(t) == UNION_TYPE;
}

static int
q_enum (tree t)
{
    return TREE_CODE(t) == ENUMERAL_TYPE;
}

static int
q_function (tree t) {
    return TREE_CODE(t) == FUNCTION_TYPE;
}

static int
```

```
q_array (tree t) {
    return TREE_CODE(t) == ARRAY_TYPE;
}

static int
q_complete( tree t ) {
    return COMPLETE_TYPE_P(t);
}

static int
q_contained( tree t ) {
    tree ctx;
    return (ctx = TYPE_CONTEXT(t))
              && (TREE_CODE(ctx) == RECORD_TYPE) ;
}

static int
q_align( tree t ) {
    if ((TREE_CODE(t) == RECORD_TYPE)
         && CLASSTYPE_AS_BASE(t))
        return CLASSTYPE_ALIGN(t);
    else
        return TYPE_ALIGN(t);
}

static int
q_bases( tree t ) {
    int cnt = -1;   /* skip over base 1: the type itself */
    if (TREE_CODE(t) != RECORD_TYPE || ! t->type.binfo)
        return 0;

    t = t->type.binfo;

    do {
        if (TREE_PUBLIC(t))
            cnt++;
    } while ((t=t->common.chain));

    return cnt;
}

static int
```

```
q_template( tree t ) {
    return (TREE_CODE(t) == RECORD_TYPE)
             && CLASSTYPE_USE_TEMPLATE(t);
}

/* returns the number of template parameters */
static int
q_templ_params( tree t ) {
    tree r;
    if (TREE_CODE(t) != RECORD_TYPE
        || !CLASSTYPE_USE_TEMPLATE(t))
        return 0;
    r = CLASSTYPE_TI_ARGS(t);
    return r->vec.length;
}

/* Counts:
 *  - static and nonstatic data members
 * Doesn't count:
 *  - enums
 */
static int
q_members( tree t ) {
    tree cur;
    int cnt = 0;
    if (TREE_CODE(t) != RECORD_TYPE)
        return 0;
    cur = TYPE_VALUES(t);
    while (cur) {
        if (is_member(cur))
            cnt++;
        cur=TREE_CHAIN(cur);
    }
    return cnt;
}

static int
q_counted_method( tree elem ) {
    char * name;
    if (!elem) return 0;

    /* eliminate methods with an initial
```

```
   double-underscore */
name = (char *) pull_name (elem);
if (name) {
    if ((name[0] == name[1]) && (name[0] == '_'))
        return 0;
}
switch (TREE_CODE(elem)) {
    case OVERLOAD: {
        /* scan for any public overloads */
        tree ovl = elem;
        while (ovl) {
            if (    !(TREE_PUBLIC (ovl)
                      || TREE_PRIVATE (ovl)
                      || TREE_PROTECTED (ovl))
                ||  ( TREE_PUBLIC (ovl)
                      &&  !(TREE_PRIVATE (ovl)
                            || TREE_PROTECTED (ovl)))
               ) {
                return 1;
                break;
            }
            ovl = TREE_CHAIN (ovl);
        }
        return 0;
    }

    case FUNCTION_DECL:
        if (    !(TREE_PUBLIC (elem)
                  || TREE_PRIVATE (elem)
                  || TREE_PROTECTED (elem))
             ||  ( TREE_PUBLIC (elem)
                   &&  !(TREE_PRIVATE (elem)
                         || TREE_PROTECTED (elem)))
            )
            return 1;
        return 0;

    case TEMPLATE_DECL:
    default:
        return 0;

}
```

```
}

static int
q_methods( tree t ) {
    tree cur;
    int len, i=0, cnt = 0;
    if (TREE_CODE(t) != RECORD_TYPE)
        return 0;
    cur = CLASSTYPE_METHOD_VEC(t);
    if (!cur)
        return 0;
    len = TREE_VEC_LENGTH(cur);
    while (i<len) {
        tree elem = TREE_VEC_ELT(cur,i);
        if (!elem) { i++; continue; }

        if (q_counted_method (elem))
            cnt++;

        i++;
    }
    return cnt;
}

/* returns the number of function parameters */
static int
q_func_params (tree t) {
    if (TREE_CODE(t) == FUNCTION_TYPE) {
        int cnt=0;
        tree cur = TYPE_ARG_TYPES(t);
        while (cur && TREE_CODE(cur) != VOID_TYPE) {
            cnt++;
            cur = TREE_CHAIN(cur);
        }
        return cnt;
    } else {
        return 0;
    }
}

static int
q_arrlen (tree tp) {
```

```
    tree dom;
    tree max;
    if (   !(TREE_CODE (tp) == ARRAY_TYPE)
        || !(dom = TYPE_DOMAIN (tp))
        || !(max = TYPE_MAX_VALUE (dom)))
        return 0;
    return TREE_INT_CST_LOW (max) + 1;
}

/*
 ** Method Enum Generators
 */
#pragma mark -+ Method Enum Generators

/* this won't be called from annotate_type_desc, as that
   only takes in types, not methods */
static int
q_trivial (tree t) {
    if (!t || TREE_CODE (t) != FUNCTION_DECL
        || !DECL_SAVED_TREE (t))
        return 0;
    /* FIXME: This code isn't supposed to work */
    t = COMPOUND_BODY (DECL_SAVED_TREE (t));
    while (t) {
        switch (TREE_CODE (t)) {
        /* put in allowed subtrees here, which replace t.
           as long as we end up with only allowed subtrees,
           the function's trivial. */
        default:
            return 0;
        }
    }
    return 1;
}

static int
q_static (tree t) {
    if (TREE_CODE (t) == OVERLOAD)
        return DECL_STATIC_FUNCTION_P (OVL_FUNCTION (t));
    else
        return DECL_STATIC_FUNCTION_P (t);
}
```

```
static int
q_const (tree t) {
    tree arg;
    int ro;

    if (TREE_CODE (t) == OVERLOAD)
        return q_const (OVL_FUNCTION (t));

    if (TREE_CODE_CLASS (TREE_CODE (t)) == 'd')
        arg = DECL_ARGUMENTS (t);
    else
        arg = 0;

    /* no params?  that's const in my book */
    if (!arg)
        return 1;

    /* look for a readonly first THIS argument */
    if (DECL_NAME (arg) == get_identifier ("this"))
        return TREE_READONLY (arg);

    /* no THIS arg, look at all of them. */
    ro = 1;
    while (arg && ro) {
        if (!TREE_READONLY (arg))
            ro = 0;
        arg = TREE_CHAIN (arg);
    }
    return ro;
}

static int
q_virtual (tree t) {
    if (TREE_CODE (t) == OVERLOAD)
        return DECL_VIRTUAL_P (OVL_FUNCTION (t));
    else
        return DECL_VIRTUAL_P (t);
}

static int
q_operator (tree t) {
```

```
    return (TREE_CODE (t) == OVERLOAD);
/*   */
/*  if (TREE_CODE_CLASS (TREE_CODE (t)) != 'd') { */
/*      printf ("non-decl in q_operator:\n"); */
/*      debug_tree (t); */
/*      return 0; */
/*  } else { */
/*      return DECL_OVERLOADED_OPERATOR_P (t); */
/*  } */
}

static int
q_nothrow (tree t) {
    return TREE_NOTHROW (t);
}

/*
 ** Utility Functions
 */
#pragma mark -- Utility Functions --
/* returns an INTEGER_CONST tree for the given integer.
   Note that some of the returned values may be
   singletons. */
/* integer_type_node */
static tree
get_integer(int n)
{
    switch (n) {
        case 0: return integer_zero_node;
        case 1: return integer_one_node;
        case 2: return integer_two_node;
        case 3: return integer_three_node;
        default: return build_int_2(n,0);
    }
}

/* returns the nth method or overload for src. idx is the
   place in the method vector.  it's value is worthless if
   this returns 0. */
static tree
nth_method (tree src, int n, int *idx) {
    tree methods, meth;
```

```
    int cnt,len,i;

    if (   !src
        || !(TREE_CODE(src) == RECORD_TYPE)
        || !(methods = CLASSTYPE_METHOD_VEC (src))
        || !(len = TREE_VEC_LENGTH (methods)))
        return 0;

    i=cnt=0;

    /* get first method */
    do {
        meth = TREE_VEC_ELT (methods, i++);
    } while (!meth && i < len);

    /* get nth method */
    while (cnt < n && i < len) {
        if (   (meth = TREE_VEC_ELT (methods, i++))
            && q_counted_method (meth))
            cnt++;
    }

    *idx = i-1;
    return meth; /* may be 0 */
}

/* returns the BINFO for the Nth public base of T */
static tree
nth_base( tree t, int n ) {
    int i,len;
    tree vec;

    if (   !t
        || !(t = TYPE_BINFO (t))
        || !(vec = BINFO_BASETYPES (t))
        || (n < 0)
        || (TREE_VEC_LENGTH (vec) <= n))
        return 0;

    i=0;
    len = TREE_VEC_LENGTH (vec);
    while (i<len) {
```

```
        tree elt = TREE_VEC_ELT (vec,i);
        if (TREE_PUBLIC (elt) && !n--)
            return elt;
    }

    return 0;
}

static int
is_member( tree t ) {
    /* public members only */
    if (TREE_PRIVATE(t) || TREE_PROTECTED(t))
        return 0;
    switch( TREE_CODE(t) ) {
        case FIELD_DECL:
        case VAR_DECL:
            return 1;
        default:
            return 0;
    }
}

static tree
pull_name_id( tree t ) {
    if (!t || t == error_mark_node)
        return 0;
    else {
        while (t && TREE_CODE (t) != IDENTIFIER_NODE) {
            switch (TREE_CODE_CLASS (TREE_CODE (t))) {
            case 't':
                t = TYPE_NAME (t);
                break;
            case 'd':
                t = DECL_NAME (t);
                break;
            default:
                t = 0;
            }
        }
        return t;
    }
}
```

```
static const char *
gen_name( tree t )
{
    if (POINTER_TYPE_P (t)) {
        char *buf;
        int len;
        const char * inner;

        inner = gen_name (TREE_TYPE (t));
        len = strlen (inner);
        buf = ggc_alloc (len + 1);
        strcpy (buf, inner);
        strcpy (buf + len, " *");
        return buf;
    }
    else
        return pull_name (t);
}

static const char *
pull_name( tree t )
{
    if ((t = pull_name_id (t)))
        return IDENTIFIER_POINTER (t);
    else
        return "";

}

static int
pull_templ_int (tree src, tree *parent, int *index) {
    tree vec, indext;
    if (   !(src
             && src != error_mark_node
             && TREE_CODE (src) != ERROR_MARK)
        || !(TREE_CODE (src) == RECORD_TYPE)
        || !(vec = CLASSTYPE_TI_ARGS (src))
        || !(TREE_VEC_LENGTH (vec) > 1)
        || !(*parent = TREE_VEC_ELT (
                        TREE_VEC_ELT (vec, 0),
                        0))
```

```
        || !(*parent != error_mark_node)
        || !(TREE_CODE (*parent) == RECORD_TYPE)
        || !(indext = TREE_VEC_ELT (
                        TREE_VEC_ELT (vec, 1),
                        0))
        || !((*index = TREE_INT_CST_LOW (indext)) >= 0))
        return 0;
     return 1;
}


/* appends a set of enums, described in ENUMS, to T.  There
should be NR_ENUMS structs pointed to by ENUMS. Ok, at least
that many. */
static tree
append_enums(tree t, struct enum_val * enums,
             int nr_enums ) {
    tree enu, cur, lbl, val, lt, ht, type;
    int i, low, high;

    lt = 0;
    ht = 0;
    low = 0;
    high = 0;

    if (nr_enums) {
        low = enums[0].value;
        high = low;
        lt = ht = NULL_TREE;
    }

    enu = make_node(ENUMERAL_TYPE);
    TYPE_CONTEXT (enu) = t;
    /*TYPE_SIZE (enu) =  int_cst <bit_size_type> (32) */
    /*TYPE_SIZE_UNIT (enu) =  int_cst <long unsigned int>
                              (4) */
    /* TREE_READONLY (enu) = 1;  TYPE_MAIN_DECL */
/*  TREE_CONSTANT (enu) = 1; */
    type = make_node (TYPE_DECL);
    TREE_TYPE (type) = enu;
    TYPE_STUB_DECL (enu) = type;
```

```
for (i=0; i<nr_enums; ++i)
{
    int cv;

    /* Generate the ID & INT_CST nodes we're
       going to use. */
    cv = enums[i].value;
    lbl = get_identifier(enums[i].name);
    val = build_int_2(cv,0);

    /* Update the range for the enumeration type being
       constructed */
    if (cv < low) {
        low = cv;
        lt = val;
    }

    if (cv > high) {
        high = cv;
        ht = val;
    }

    /* build the CONST_DECL for the type */
    cur = build_decl(CONST_DECL, lbl, enu);
    TREE_READONLY (cur) = 1;
    TREE_CONSTANT (cur) = 1;
    DECL_INITIAL(cur) = val; /*convert(enu, val); */
    DECL_NAME(cur) = lbl;

    /* Add the label & value to the enum's type */
    TYPE_VALUES(enu) = tree_cons(lbl, val,
                                 TYPE_VALUES(enu));

    /* add the CONST_DEC to t */
    TREE_CHAIN(cur) = TYPE_VALUES(t);
    TYPE_VALUES (t) = cur;
}

/*TYPE_VALUES(t) = nreverse(cur); */

/*
 Copied from decl.c:finish_enum
```

```
     */
    for (cur = TYPE_VALUES (enu);
         cur;
         cur = TREE_CHAIN (cur))
        TREE_TYPE (TREE_VALUE (cur)) = enu;

    /*TYPE_PRECISION (enu) = TYPE_PRECISION (
                                 integer_type_node
                            ); */
    TYPE_MODE (enu) = TYPE_MODE (integer_type_node);

    TYPE_MIN_VALUE(enu) = lt? lt : get_integer(low);
    TYPE_MAX_VALUE(enu) = ht? ht : get_integer(high);

    return t;
}

static int
lookup_enum( tree src, const char * name )
{
    int enu = 0;
    tree nm = get_identifier (name);
    tree cur = TYPE_VALUES (src);

    while (cur && TYPE_NAME(cur) != nm)
        cur = TREE_CHAIN (cur);

    if (cur && TREE_CODE(cur) == CONST_DECL)
        enu = TREE_INT_CST_LOW (DECL_INITIAL (cur));

    return enu;
}

/* appends a typedef to DEST, with name NAME and
   type TYPE */
static tree
append_typedef( tree dest, const char * name, tree type ) {
    tree decl, idname, typecpy;

    idname = get_identifier (name);

    /* duplicate the underlying type */
```

```
    typecpy = build_type_copy (type);

    /* copy over debug info, so only 1's generated for all
       typedefs */
    TYPE_STUB_DECL (typecpy) = TYPE_STUB_DECL (type);

    /* build a decl node */
    decl = build_lang_decl (TYPE_DECL, idname, typecpy);
    TYPE_NAME (typecpy) = decl;
    TREE_TYPE (decl) = typecpy;
    DECL_ORIGINAL_TYPE (decl) = type;
    TREE_USED (typecpy) = TREE_USED (decl);

    DECL_NONLOCAL (decl) = 1;
    TREE_PRIVATE (decl) = 0;
    TREE_PROTECTED (decl) = 0;
    TREE_PUBLIC (decl) = 1;

    /* add it to the member list */
    DECL_CONTEXT (decl) = dest;
    TREE_CHAIN (decl) = TYPE_FIELDS (dest);
    TYPE_FIELDS (dest) = decl;

    return dest;
}

static tree
append_string( tree dest, const char * name,
               const char * value ) {
    tree initval;
    tree constchar;

    /* the tree constructed here is a near-duplicate of what
       I found of the tree an initialized declaration in
       code generates.  Probably easier ways to do this with
       simpler trees */

    constchar = copy_node (char_type_node);
    TYPE_POINTER_TO (constchar) = 0;
    TYPE_REFERENCE_TO (constchar) = 0;
    TYPE_ATTRIBUTES (constchar) = 0;
    TYPE_MAIN_VARIANT (constchar) = char_type_node;
```

```
TYPE_READONLY (constchar) = 1;

/* construct the initial value of the decl */
{
    tree str_cst;
    tree addr_expr;
    int len;
    char * strd;

    /* copy the string & zero-terminate the copy */
    len = strlen(value);
    strd = ggc_alloc(len+1);
    memcpy(strd,value,len);
    strd[len]=0;

    /* make a STRING_CST for the copy */
    str_cst = build_string( len+1, strd );
    str_cst = fix_string_type( str_cst );
    TREE_READONLY (str_cst) = 1;
    TREE_CONSTANT (str_cst) = 1;
    TREE_STATIC (str_cst) = 1;

    /* set up an ADDR_EXPR to the constant */
    addr_expr = build_address( str_cst );

    /* and a NOP_EXPR that gets the address */
    initval= build_nop( build_pointer_type( constchar ),
                        addr_expr );

}

/* construct the FIELD_DECL */
{
    tree typep;
    tree idname;

    /* const pointer to const char */
    typep = build_pointer_type (constchar);
    TYPE_READONLY (typep) = 1;

    /* the name*/
    idname = get_identifier(name);
```

```
        /* put in the FIELD_DECL */
        append_decl (dest, idname, typep, initval);

    }

    return dest;
}

static void
append_decl (tree dest, tree name, tree type,
             tree initial) {
    tree decl;

    decl = build_lang_decl (VAR_DECL, name, type);
    TREE_STATIC (decl) = 1;
    DECL_INITIAL (decl) = initial;
    DECL_INITIALIZED_P (decl) = 1;
    /* DECL_INITIALIZED_BY_CONSTANT_EXPRESSION_P (decl) = 1;
    */
    DECL_INITIALIZED_IN_CLASS_P (decl) = 1;

    TREE_PUBLIC (decl) = 1;
    TREE_PRIVATE (decl) = 0;
    TREE_PROTECTED (decl) = 0;
    TREE_READONLY (decl) = 1;

    DECL_EXTERNAL (decl) = 1;
    DECL_NONLOCAL (decl) = 1;
    DECL_IN_AGGR_P (decl) = 1;
    DECL_THIS_STATIC (decl) = 1;

    /* add it to dest */
    DECL_CONTEXT (decl) = dest;
    TREE_CHAIN (decl) = TYPE_FIELDS (dest);
    TYPE_FIELDS (dest) = decl;
}

/*
 ** Annotation Drivers
 */
#pragma mark --Annotation Drivers--
```

```
typedef int (*query_func)(tree);
static struct _unused_1{
    const char * name;
    query_func   func;
} enums[] = {
    /*
     * Category 1: Basic Classification
     */
    { "is_class", q_class },
    { "is_union", q_union },
    { "is_enum", q_enum },
    { "is_function", q_function },
    { "is_array", q_array },
    { "is_pod", pod_type_p },
    { "align", q_align },

    /*
     * Category 2: Type Browsing
     */
    { "is_complete", q_complete },
    /* That's not available to us here. */
    /* { "is_typedef", q_typedef }, */
    { "is_contained", q_contained },
    /* This one won't be added to type_description types. */
    /* { "is_trivial", q_trivial }, */
    { "num_bases",  q_bases },
    { "is_template", q_template },
    { "num_templ_parameters", q_templ_params },
    { "num_params", q_func_params },
    { "num_members", q_members },
    { "num_methods", q_methods },
    { "array_length", q_arrlen }
};
#define NR(x)  (sizeof(x)/sizeof(x[0]))

static tree
annotate_type_desc(tree td)
{
    tree param;
    const char * name;
    struct enum_val enum_vals[NR(enums)];
```

```
    unsigned i;

    /* get our template parameter */
    param = CLASSTYPE_TI_ARGS(td);
    if (!param || TREE_VEC_LENGTH(param) < 1)
    {
        /* we need a parameter in here! */
        return error_mark_node;
    }

    param = TREE_VEC_ELT(param, 0);

    for (i=0; i<NR(enums); ++i)
    {
        enum_vals[i].name = enums[i].name;
        enum_vals[i].value = (*enums[i].func)(param);
    }

    append_enums (td, enum_vals, NR(enum_vals) );

    name = gen_name (original_type (param));

    append_string (td, "name",  name);

    if (q_contained(param))
        append_typedef (td, "container_type",
                        TYPE_CONTEXT (param) );

    if (q_array(param))
        append_typedef (td, "inner_type",
                        TREE_TYPE (param));

    if (q_function(param))
        append_typedef (td, "return_type",
                        TREE_TYPE (param));

    return td;
}

static tree
annotate_baseclass (tree bc) {
    tree parent, base;
```

```
    int bcnt;
    struct enum_val is_virtual;

    /* get the class and the base class number */
    if (!pull_templ_int (bc, &parent, &bcnt))
        return bc;

    base = nth_base( parent, bcnt );
    /* From here, we need to add in the enums and typedefs
       required. */

    if (!base)
        return bc;

    is_virtual.name = "is_virtual";
    is_virtual.value = TREE_VIA_VIRTUAL(base);
    append_enums( bc, &is_virtual, 1 );

    append_typedef( bc, "base_type", BINFO_TYPE(base) );
    return bc;
}

static tree
annotate_templparam (tree tp) {
    tree parent, param, tparams;
    int tcnt;
    const char * ptype;
    struct enum_val param_type;

    /* get the class and the template parameter number */
    if (!pull_templ_int (tp, &parent, &tcnt))
        return tp;

    tparams = CLASSTYPE_TI_ARGS(parent);
    if (   !tparams
        || !(TREE_VEC_LENGTH(tparams) >tcnt)
        || !(param = TREE_VEC_ELT(tparams, tcnt)))
        return tp;

    /* Part 1: the instantiated type */
    switch (TREE_CODE (param))
      {
```

```
    case INTEGER_CST:
        ptype = "tp_int";
        param_type.name = "value_cst";
        param_type.value = TREE_INT_CST_LOW (param);
        append_enums (tp, &param_type, 1);
        break;

    default:
        ptype = "tp_class";
        append_typedef(tp, "value_type", param);

        /* Part 2: the name, which isn't relevant for int
                   params. */
        append_string (tp, "name",  pull_name (param));

      }

    /* Part 3: The parameter's formal type */
    param_type.name = "param_type";
    param_type.value = lookup_enum(parent, ptype);
    append_enums (tp, &param_type, 1);

    return tp;
}

static tree
annotate_param (tree tp){
    tree parent, args;
    int cnt;

    if (!pull_templ_int (tp, &parent, &cnt))
        return tp;

    args = DECL_ARGUMENTS (parent);

    while (args && cnt--)
        args = TREE_CHAIN(args);

    if (args)
      {
        append_typedef(tp, "value_type",
                       TREE_VALUE (args) );
```

```
        /* function types can't have default values! */
      }

    return tp;
}
/* TYPE_FIELDS */
/* Common function for both annotate_method and
   annotate_method_overload */
static tree
annotate_any_method (tree dest, tree src, tree unused) {
    tree ptrt, ptr_init;
    int i;
    unused = 0;

    if (TREE_CODE (src) == ERROR_MARK)
        return dest;

    struct enum_val enums[] = {
        {"is_const",0},
        {"is_virtual",0},
        {"is_static",0},
        {"is_trivial",0},
        {"is_nothrow",0}
    };

    /* I just hate hardcoding... */
    i=0;
    enums[i++].value = q_const (src);
    enums[i++].value = q_virtual (src);
    enums[i++].value = q_static (src);
    enums[i++].value = q_trivial (src);
    enums[i++].value = q_nothrow (src);
    append_enums (dest, enums, 5);

    append_typedef (dest, "type", TREE_TYPE (src));

    ptrt = build_pointer_type (TREE_TYPE (src));
    ptr_init = build_nop (ptrt, build_address (src));
    append_decl (dest, get_identifier ("ptr"), ptrt,
                 ptr_init);

    return dest;
```

```
}

static tree
annotate_method (tree tp) {
    tree parent, meth, func;
    int index, overloads=0;
    int dtor = 0;
    int real_idx;
    int mode = 0;

    struct enum_val enums[] = {
        {"num_overloads",0},
        {"is_ctor",0},
        {"is_dtor",0},
        {"is_operator",0}
    };

    if (!pull_templ_int (tp, &parent, &index))
        return tp;

    meth = nth_method (parent, index, &real_idx);

    /* handle overloads specially */
    if (TREE_CODE (meth) == OVERLOAD) {
        tree t;
        func = OVL_FUNCTION (meth);
        /* just count the overloads */
        t = meth;
        while (t) {
            overloads++;
            t = TREE_CHAIN (t);
        }
        enums[0].value = overloads;
    } else {
        /* enums[0] stays = 0 */
        annotate_any_method (tp, meth, parent);
        func = meth;
    }

    /* put in the normal enum vals */
    enums[1].value = (real_idx == 0);
    enums[2].value = dtor = (real_idx == 1);
```

```
    enums[3].value = q_operator (meth);

    append_enums(tp, enums, 4);

    if (            enums[1].value
        || (++mode, enums[2].value)
        || (++mode, enums[3].value) ) {
        char * nm=0;
        switch (mode) {
        case 0: /* ctor */
            nm = (char *) pull_name (parent);
            break;
        case 1: /* dtor */ {
            tree pname;
            pname = pull_name_id (parent);
            if (!pname)
                pname = get_identifier ("");
            nm = ggc_alloc (IDENTIFIER_LENGTH (pname) + 2);
            nm[0] = '~';
            memcpy (nm+1, IDENTIFIER_POINTER (pname),
                          IDENTIFIER_LENGTH (pname));
            nm[IDENTIFIER_LENGTH(pname)+1] = 0;
            break;
        }
        case 2: /* operator */
            if (real_idx == 2) { /* conversion operators */
                nm = (char *) "operator __convert";
                  /* the type will have to be deduced
from the args */
            } else {
                /* build a full name for this operator */
                nm = (char *) pull_name (meth);
            }
            break;
        }

        append_string (tp, "name", nm);
    } else {
        append_string (tp, "name", pull_name (meth));
    }

    return tp;
```

```
}

static tree
annotate_method_overload (tree src) {
    tree vec, parent, indexmp, indexop,
         meth, ovl;
    int indexm, indexo, real_index; /* index-method,
                                       index-overload,
                                       real-index */

    /* this thing's a big block of ugly, but it's not too
       bad once you get used to the syntax style */
    if (   !(src && src != error_mark_node)
        || !(TREE_CODE (src) == RECORD_TYPE)

        || !(vec = CLASSTYPE_TI_ARGS (src))
        || !(TREE_VEC_LENGTH (vec) > 2)

        || !(parent = TREE_VEC_ELT (
                        TREE_VEC_ELT (vec, 0),
                        0))
        || !(parent != error_mark_node)
        || !(TREE_CODE (parent) == RECORD_TYPE)

        /* get the method index */
        || !(indexmp = TREE_VEC_ELT (
                        TREE_VEC_ELT (vec, 1),
                        0))
        || !((indexm = TREE_INT_CST_LOW (indexmp)) >= 0)

        /* get the overload index */
        || !(indexop = TREE_VEC_ELT (
                        TREE_VEC_ELT (vec, 2),
                        0))
        || !((indexo = TREE_INT_CST_LOW (indexop)) >= 0)

        || !(meth = nth_method (parent, indexm,
                                &real_index))
        || !(TREE_CODE (meth) == OVERLOAD)) {
        return src;
    }
```

```
    ovl = meth;
    while (indexo && ovl) {
        indexo--;
        ovl = TREE_CHAIN (ovl);
    }

    if (ovl)
        annotate_any_method (src, ovl, parent);

    return src;
}

static struct enum_val am_enums[] = {{"is_static", 0}};

static tree
annotate_member (tree tp) {
    tree parent, mem;
    tree ptr_init, ptrt, type;
    int n,cnt;

    if (!pull_templ_int (tp, &parent, &n))
        return tp;

    cnt=n;

    if (parent == error_mark_node)
        return tp;

    mem = TYPE_FIELDS (parent);

    /* skip to first member for cnt=0 */
    while (mem && !is_member (mem))
        mem = TREE_CHAIN (mem);

    /* keep going for cnt > 0 */
    while (cnt && mem) {
        if (is_member (mem))
            --cnt;
        mem = TREE_CHAIN (mem);
    }

    if (!mem) {
```

```
        /* Silent error handling - empty definition! */
        /* error ("there is no public member \%d of \%T",
                  n, parent); */
        return tp;
    }

    /* add in type information about mem to tp */
    append_typedef (tp, "type", type=TREE_TYPE (mem));
    if (DECL_NAME (mem)) {
        append_string (tp, "name",
                       IDENTIFIER_POINTER (DECL_NAME (mem)));
    } else {
        append_string (tp, "name", "__unnamed");
    }

    if ((am_enums[0].value = TREE_STATIC (mem))) {
        /* add pointer-to-static-member here */
        ptr_init = build_address( mem );
        type = TREE_TYPE (mem);
        ptrt = TYPE_POINTER_TO (type);
    } else {
        /* add a pointer-to-member member here */
        ptrt = build_pointer_type (
                  build_offset_type (parent, type));
        ptr_init = make_ptrmem_cst (ptrt, mem);
    }

    ptr_init = build_nop (ptrt, ptr_init);
    append_decl (tp, get_identifier ("ptr"),
                 ptrt, ptr_init);
    append_enums (tp, am_enums, 1);

    return tp;
}

#define DONT_ANNOTATE       0
#define ANNOTATE_TYPEDESC   1
#define ANNOTATE_PARAM      2
#define ANNOTATE_BASECLASS  3
#define ANNOTATE_TEMPLPARAM 4
#define ANNOTATE_MEMBER     5
#define ANNOTATE_METHOD     6
```

```
#define ANNOTATE_METH_OVERLOAD  7

/* Returns DONT_ANNOTATE
             if this isn't a CTTI annotated class,
           ANNOTATE_TYPEDESC
             if this is std::type_description
           ANNOTATE_PARAM
             if this is std::type_description::param
           ANNOTATE_BASECLASS
             if this is std::type_description::base_class
           ANNOTATE_TEMPLPARAM
             if this is std::type_description
                           ::template_param
           ANNOTATE_MEMBER
             if this is std::type_description::member
           ANNOTATE_METHOD
             if this is std::type_description::method
           ANNOTATE_METH_OVERLOAD
             if this is a std::type_description
                          ::method::overload
           */
static int
should_annotate(tree tr)
{
    tree name;
    tree ctx;

    if (!tr || TREE_CODE(tr) != RECORD_TYPE)
        return 0;

    if (TREE_CODE (tr) == ERROR_MARK)
        return 0;

    /* look for std::type_description */
    tr = CLASSTYPE_TEMPLATE_INFO(tr);
    if (!tr) return 0;

    tr = TREE_PURPOSE(tr);
    if (!tr) return 0;

    /* match the namespace as "std" */
    if (DECL_CONTEXT(tr) == std_node)
```

```
{
  /* match type_description */
  if (!(name = DECL_NAME(tr)))
      return 0;

  if (strcmp(IDENTIFIER_POINTER(name),
             "type_description"))
      return 0;

  /* this is it */
  return ANNOTATE_TYPEDESC;
} else if (   (ctx=DECL_CONTEXT(tr))
           && (TREE_CODE(ctx) == RECORD_TYPE)) {
  int ctx_tp = should_annotate (ctx);
  if (ctx_tp == ANNOTATE_TYPEDESC) {
      /* try looking for
          type_description-embedded classes */
      const char * idp;

      if (   !(name = DECL_NAME(tr))
          || !(idp = IDENTIFIER_POINTER(name)) )
          return 0;

      /* these can actually just be pointer
         comparisons against
         the unique identifier_nodes */

      if (!strcmp(idp, "param"))
          return ANNOTATE_PARAM;

      if (!strcmp(idp, "base_class"))
          return ANNOTATE_BASECLASS;

      if (!strcmp(idp, "template_param"))
          return ANNOTATE_TEMPLPARAM;

      if (!strcmp(idp, "member"))
          return ANNOTATE_MEMBER;

      if (!strcmp(idp, "method"))
          return ANNOTATE_METHOD;
```

```
          } else if (ctx_tp == ANNOTATE_METHOD) {
            const char * idp;
            if (   !(name = DECL_NAME (tr))
                || !(idp = IDENTIFIER_POINTER (name)) )
                return 0;

            if (!strcmp (idp, "overload"))
                return ANNOTATE_METH_OVERLOAD;

          }
    }

    return 0;

}

/*
 ** Our exported API
 */

/* should be called from:
    instantiate_class_template

    ??
    instantiate_template
    instantiate_type
    instantiate_decl
    instantiate_pending_templates
*/
tree
ctti_maybe_annotate(tree tr)
{
    switch (should_annotate(tr)) {
        case ANNOTATE_TYPEDESC:
            return annotate_type_desc (tr);

        case ANNOTATE_BASECLASS:
            return annotate_baseclass (tr);

        case ANNOTATE_TEMPLPARAM:
            return annotate_templparam (tr);
```

```
        case ANNOTATE_PARAM:
            return annotate_param (tr);

        case ANNOTATE_MEMBER:
            return annotate_member (tr);

        case ANNOTATE_METHOD:
            return annotate_method (tr);

        case ANNOTATE_METH_OVERLOAD:
            return annotate_method_overload (tr);

        case DONT_ANNOTATE:
        default:
            return tr;
    }
}
```

Appendix B

Case Studies Source

B.1 Serialization Example Source

B.1.1 serializer.h

```
#ifndef INCLUDE_SERIALIZER_H
#define INCLUDE_SERIALIZER_H
#include <string.h>
#include <stdio.h>
#include <ctti>
#include "tuple.h"
#include "bytestream.h"

template<class T> struct SerializeInfo;

// works for any POD type.
template<class T> struct PrimitiveSerializer {
    static void put (const T& value, bytestream& dest) {
        printf ("serializing type %s\n",
            std::type_description<T>::name);
        dest << value;
    }

    static T    get (bytestream& src) {
        T result;
        src >> result;
        return result;
    }
```

```
};

template<> struct PrimitiveSerializer<const char*> {
    static void put (const char *value, bytestream& dest) {
        unsigned int len;
        printf ("putting \"\%s\"\n", value);
        if (!value || !*value) {
            len = 0;
            dest << len;
        } else {
            len = strlen (value);
            dest << len;
            while (len--)
                dest << *value++;
        }
    }
    static const char * get (bytestream& src) {
        unsigned int len;
        src >> len;
        if (len) {
            char *p, *dest = (char*) malloc (len+1);
            p = dest;
            len++; // terminating zero!
            while (len--)
                src >> *p++;
            return dest;
        } else {
            return 0;
        }
    }
};

template<class T, int N> struct SerializeWorker {
    static void put (const T& value, bytestream& dest) {
        typedef typename std::type_description<T>::template member<N-1>
            member_info;
        typedef typename SerializeInfo<
                            typename member_info::type
                        >::serializer
            serializer;

        printf ("\%s[\%d]: putting member \%s of type \"\%s\"\n",
```

```
            std::type_description<T>::name,
            N,
            member_info::name,
            std::type_description<typename member_info::type>::name);

        serializer::put (value.*(member_info::ptr),dest);
        SerializeWorker<T,N-1>::put (value, dest);
    }

    static void get (T& dest, bytestream& src) {
        typedef typename std::type_description<T>::template member<N-1>
            member_info;
        src.*member_info::ptr
            = SerializeInfo<typename member_info::type>::serializer::get (
                  dest
              );
        SerializeWorker<T,N-1>::get (dest, src);
    }
};

template<class T> struct SerializeWorker<T,0> {
    static void put (const T& value, bytestream& dest) {}
    static void get (T& dest, bytestream& src) {}
};

template<class T> struct Serializer {
    static void put (const T& value, bytestream& dest) {
        printf ("Decomposing type \%s\n", std::type_description<T>::name);
        SerializeWorker<T, std::type_description<T>::num_members>::put (
            value,
            dest
        );
    }

    static T  get (bytestream& src) {
        T result;
        SerializeWorker<T, std::type_description<T>::num_members>::get (
            result,
            src
        );
        return result;
    }
```

```
};

// Determines the proper Serializer type. Override as needed.
template<class T>
struct SerializeInfo {
    typedef typename tuple::select<
                        PrimitiveSerializer<T>,
                        Serializer<T>,
                        std::type_description<T>::is_pod
                    >::result
                serializer;
};

// we provide no typedef for serializer for pointer types.
template<class T>
struct SerializeInfo<T*> {};

// we provide no typedef for serializer for reference types.
template<class T>
struct SerializeInfo<T&> {};

template<>
struct SerializeInfo<const char*> {
    typedef PrimitiveSerializer<const char*> serializer;
};

// override table.
#define DEF_ALIAS_SERIALIZER(ALIAS,TYPE) template<> struct \
SerializeInfo<ALIAS> { typedef SerializeInfo<TYPE>::serializer \
serializer; }

DEF_ALIAS_SERIALIZER (char *, const char *);


#endif
```

B.1.2 doserialize.cpp

This is the test driver for the serialization mechanism.

```
#include "bytestream.h"
#include "serializer.h"
#include <stdio.h>
struct data {
    int ival;
    char *sval;
    data () { ival = 1; sval = "data"; }
};

int main (int args, char **argv) {
    data d;
    d.ival = 0xdeadbeef;
    d.sval = "foo";
    bytestream bs ("foo.out");
    SerializeInfo<data>::serializer::put (d,bs);
    puts ("writing to foo.out");
    return 0;
}
```

B.1.3 tuple.h

```
// tuple.h
#ifndef INCLUDE_TUPLE_H
#define INCLUDE_TUPLE_H
namespace tuple {
    template<class Head, class Tail> struct T {
        typedef Head head;
        typedef Tail tail;
    };

    struct Term {};

    template<class Tup, int N> struct get {
        typedef get<typename Tup::tail,N-1> value;
    };

    template<class Tup> struct get<Tup,1> {
        typedef typename Tup::head value;
    };

    template<class Tup> struct length {
```

```
        enum { value = 1 + length<typename Tup::tail>::value };
    };

    template<> struct length<Term> {
        enum { value = 0 };
    };



    template<class A, class B> struct compare {
        enum { value = 0 };
    };

    template<class A> struct compare<A, A> {
        enum { value = 1 };
    };


    template<class Tup, class F> struct find {
        enum { value = compare<typename Tup::head, F>::value?
                            0: 1 + find<typename Tup::tail, F>::value };
    };

    template<class F> struct find<Term,F> {
        enum { value = 0 };
    };

    //
    // if N, return A, else return B.
    template<class True, class False, int N> struct select {
        typedef True result;
    };

    template<class True, class False> struct select<True,False,0> {
        typedef False result;
    };

} // end namespace tuple.
#endif // #ifndef INCLUDE_TUPLE_H
```

B.1.4 bytestream.h

A utility module used by the serializer.

```
#ifndef INCLUDE_BYTESTREAM_H
#define INCLUDE_BYTESTREAM_H

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

class bytestream {
    int fd;
public:
    bytestream (const char *fname)
      : fd (::open(fname, O_RDWR|O_CREAT, 0600)) {
        if (fd < 0) { puts (strerror (errno)); exit (1); }
    }
    ~bytestream () { ::close (fd); }

    void read (void *dest, int sz) {
        ::read (fd, dest, sz);
    }

    void write (void *src, int sz) {
        ::write (fd, src, sz);
    }
};

template<class T>
bytestream& operator << (bytestream & dest, T& o) {
    dest.write ((void*) &o, sizeof (T));
    return dest;
}

template<class T>
bytestream& operator >> (bytestream& src, T& o) {
    src.read ((void*)&o, sizeof (T));
    return src;
```

```
}

#endif
```

B.2 Container Example

Below we present the full source code to the example container.

B.2.1 vector.h

The optimized container declaration.

```
#ifndef INCLUDE_VECTOR_H
#define INCLUDE_VECTOR_H

#include <string.h>
#include <stdlib.h>
#include <ctti>
#include <new>

template<class T, int N>
struct vector_ops {
    void copy (T* dest, T* src_begin, T* src_end) {
        ::memcpy (dest, src_begin, src_end - src_begin );
    }

    void destroy (T* src_begin, T* src_end) {}
};

template<class T>
struct vector_ops<T,0> {
    void copy (T* dest, T* src_begin, T* src_end) {
        while (src_begin != src_end) {
            new (dest) T(*src_begin);
            src_begin++;
            dest++;
        }
    }
```

```
    void destroy (T* src_begin, T* src_end) {
        while (src_begin != src_end) {
            src_begin->~T();
            src_begin++;
        }
    }
};

template<class T>
class vector : private vector_ops<T, std::type_description<T>::is_pod> {
    T   *storage;
    int capacity, size;
public:
    vector () : storage(0), capacity (0), size (0) {}
    ~vector () { delete storage; }
    void push_back (const T& value) {
        if (size == capacity) {
            T * new_store;
            capacity = (capacity?
                  (capacity + capacity/2) : 16 );
            new_store = (T*) malloc (sizeof (T) * capacity);
            copy (new_store, storage, storage+size);
            destroy (storage, storage+size);
            delete storage;
            storage = new_store;
        }
        new (storage + size++) T(value);
    }
};

#endif
```

B.2.2 docontainer.cpp

Test file for optimized container.

```
#include "vector.h"
#include <sys/time.h>
#include <stdio.h>
#include <string.h>
```

```
struct yes_pod {
    int val1;
    char val2;
};

struct not_pod {
    int val1;
    char val2;
    not_pod () : val1(0), val2(0) {}
};

void show_result (char *prefix,
                  struct timeval& start,
                  struct timeval& end) {
    // implement borrowing correctly
    if (end.tv_usec - start.tv_usec < 0) {
        end.tv_usec += 1000000;
        end.tv_sec -= 1;
    }

    long sec, usec;
    sec = end.tv_sec - start.tv_sec;
    usec = end.tv_usec - start.tv_usec;

    double result = sec + (usec / 1000000.0);

    printf ("\%s: \%f seconds\n",prefix, result);
}

int main (int args, char ** argv) {

    struct timeval  start, end;
    int nr_iter;

    if (args > 1)
        nr_iter = atoi (argv[1]);
    else
        nr_iter = 10000;

    yes_pod sample_yes;
    sample_yes.val1 = 3;
    sample_yes.val2 = 'C';
```

```
    not_pod sample_no;

    // by putting the vectors in braces, we call their destructors
    // at the closing brace.  When using large iteration counts, this
    // really helps.
    {
        vector<yes_pod> ypv;

        gettimeofday (&start, 0);
        for (int i=0; i<nr_iter; i++)
            ypv.push_back(sample_yes);
        gettimeofday (&end, 0);

        show_result ("yes_pod", start, end);

    }

    {
        vector<not_pod>  npv;

        gettimeofday (&start, 0);
        for (int i=0; i<nr_iter; i++)
            npv.push_back(sample_no);
        gettimeofday (&end, 0);

        show_result ("not_pod", start, end);

    }
    return 0;
}
```

B.3 Runtime Example

Below we present the full source code to the runtime example.

B.3.1 rttd.h

Core of the run–time introspection example.

```
#ifndef INCLUDE_RTTD_H
#define INCLUDE_RTTD_H

#include <ctti>

struct class_info;
struct method_info;
struct member_info;

//
//  Class, method, and member descriptions.  For a fairly silly reason*,
// it's only possible to generate linked-lists automatically instead of
// arrays.
//
// * No way to concatenate static initializers for arrays.

struct class_info {
    const char * name;
    int nr_methods;
    int nr_members;
    method_info  *methods;
    member_info  *members;
};

struct method_info {
    const char * name;
    method_info *next;
};

struct member_info {
    const char * name;
    const char * type;
    member_info *next;
};

template<class T, int N> struct method_gen : public method_info {
    typedef typename std::type_description<T> typedesc;
    typedef typename typedesc::method<N-1> method_desc;
    method_gen<T,N-1> next_meth;
    method_gen () {
        name = method_desc::name;
        next  = & next_meth;
```

```
    }
};

template<class T> struct method_gen<T,1> : public method_info {
    typedef typename std::type_description<T> typedesc;
    typedef typename typedesc::method<0> method_desc;
    method_gen () {
        name = method_desc::name;
        next = 0;
    }
};

template<class T, int N> struct member_gen : public member_info {
    typedef typename std::type_description<T> typedesc;
    typedef typename typedesc::member<N-1> member_desc;
    member_gen<T,N-1> next_mem;
    member_gen () {
        name = member_desc::name;
        type = std::type_description< typename member_desc::type >::name;
        next = & next_mem;
    }
};

template<class T> struct member_gen<T,1> : public member_info {
    typedef typename std::type_description<T> typedesc;
    typedef typename typedesc::member<0> member_desc;
    member_gen () {
        name = member_desc::name;
        type = std::type_description< typename member_desc::type >::name;
        next = 0;
    }
};


template<class T> struct introspective_generator {
    typedef typename std::type_description<T> typedesc;
    struct class_init : public class_info {
        method_gen<T, typedesc::num_methods> gen_methods;
        member_gen<T, typedesc::num_members> gen_members;
        class_init () {
            name =typedesc::name;
            nr_methods =typedesc::num_methods;
```

```
            nr_members =typedesc::num_members;
            methods = & gen_methods;
            members = & gen_members;
        }
    } generated_class_info;
};

#endif
```

B.3.2 introspect.cpp

Test driver for the run–time introspection example.

```
#include "rttd.h"
#include <stdio.h>

struct introspect_me {
    int  member_one;
    int  member_two;
    void method_one () {}
    void method_two () {}
};

void print_member (member_info *m) {
    if (!m) return;
    printf ("\tMember: \"\%s\": \%s\n",
             m->name,
             m->type);
    print_member (m->next);
}

void print_method (method_info *m) {
    if (!m) return;
    printf ("\tMethod: \"\%s\"\n",
            m->name);
    print_method (m->next);
}

int main (int args, char ** argv) {
    static introspective_generator<introspect_me> gen;
```

```
    class_info *ci = & gen.generated_class_info;

    printf ("Class \"\%s\"\n", ci->name);
    print_member (ci->members);
    print_method (ci->methods);
    return 0;
}
```

Bibliography

[1] Apple Human Interface Guidelines. `http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/OSXHIGuidelines.pdf`, 2004.

[2] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[3] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *The ANSI C++ Standard (ISO/IEC 14882)*, 2002.

[4] Gordon S. Blair, Geoff Coulson, Lynne Blair, Hector Duran-Limon, Paul Grace, Rui Moreira, and Nikos Parlavantzas. Reflection, Self-Awareness and Self-Healing in OpenORB. In *Proceedings of the First Workshop on Self-Healing Systems*, pages 9–14. ACM Press, 2002.

[5] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. *SIGPLAN Not.*, 23(SI):1–142, 1988.

[6] Fabian Breg and Constantine D. Polychronopoulos. Java Virtual Machine Support for Object Serialization. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, pages 173–180. ACM Press, 2001.

[7] J.-P. Briot and P. Cointe. Programming with Explicit Metaclasses in Smalltalk-80. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 419–431. ACM Press, 1989.

[8] Licia Capra, Gordon S. Blair, Cecilia Mascolo, Wolfgang Emmerich, and Paul Grace. Exploiting Reflection in Mobile Computing Middleware. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):34–44, 2002.

[9] Shigeru Chiba. OpenC++ Home Page. `http://www.csg.is.titech.ac.jp/~chiba/openC++.html`.

[10] Shigeru Chiba. OpenC++ Tutorial. `http://www.csg.is.titech.ac.jp/~chiba/opencxx/tutorial.pdf`.

[11] Shigeru Chiba. A Metaobject Protocol for C++. In *Conference on Object Oriented Programming Systems Languages and Applications*. SIGPLAN: ACM, ACM, 1995.

[12] Tyng-Ruey Chuang, Y. S. Kuo, and Chien-Min Wang. Non-Intrusive Object Introspection in C++: Architecture and Application. In *Proceedings of the 20th International Conference on Software Engineering*, pages 312–321. IEEE Computer Society, 1998.

[13] Apple Computer. Cocoa: NSProxy Class (Objective-C). `http://developer.apple.com/documentation/Cocoa/Reference/Foundation/ObjC_classic/Classes/NSProxy.html`.

[14] Apple Computer. Cocoa: The Objective-C Programming Language. `http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/index.html`.

[15] Apple Computer. WebObjects: Enterprise Key Concepts of Object-Oriented Programming. `http://developer.apple.com/documentation/WebObjects/Enterprise_Objects/Introduction/chapter_2_section_6.html`.

[16] Apple Computer. WebObjects Overview. `http://developer.apple.com/documentation/WebObjects/WebObjects_Overview/WebObjects_Overview.pdf`.

[17] Apple Computer. Cocoa. `http://developer.apple.com/cocoa/`, 2003.

[18] Comeau Computing. C++ Compilers From Comeau Computing Targeting ANSI/ISO C++ standard / C and C++ Compiler for Multi-Platform Needs. `http://www.comeaucomputing.com`.

[19] Jonathan Corbet. LWN - Kernel. `http://lwn.net/2000/1109/kernel.php3`.

[20] Metrowerks Corporation. Codewarrior Development Studio. `http://www.metrowerks.com/MW/Develop/compiler.htm`.

[21] Metrowerks Corporation. Codewarrior Pro 8 for the Macintosh, 2003.

[22] Microsoft Corporation. Visual C++: Adding Functionality. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/vcovrProgrammingTechniques.asp`.

[23] Microsoft Corporation. Visual C++ Developer Center. `http://msdn.microsoft.com/visualc/`.

[24] Microsoft Corporation. Visual C++ .NET 2003 Enhanced Compiler Conformance. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclang/html/vclrfvisualcnet2003enhancedcompilerconformance.asp`.

[25] Steven Dekorte. Objective-C. `http://www.dekorte.com/Objective-C/index.html`.

[26] Distributed Object Computing Group. TAO Overview. http://www.cs.wustl.edu/~schmidt/TAO-intro.html.

[27] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[28] B. Foote and R. E. Johnson. Reflective Facilities in Smalltalk-80. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 327–335. ACM Press, 1989.

[29] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. Clos: Integrating Object-Oriented and Functional Programming. *Commun. ACM*, 34(9):29–38, 1991.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[31] The GNUstep Project. GNUstep.org. http://www.gnustep.org/.

[32] Ira P. Goldstein and Daniel G. Bobrow. Extending Object Oriented Programming in Smalltalk. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, pages 75–81. ACM Press, 1980.

[33] Brendan Gowing and Vinny Cahill. Meta-Object Protocols for C++: The Iguana Approach. In *Proceedings of the Reflection '96 Conference*, San Francisco, California, USA, 1996. Xerox Palo Alto Research Center.

[34] Duncan Grisby. omniORB: Free High Performance ORB. http://omniorb.sourceforge.net.

[35] Object Management Group. *CORBA 2.6 - Chapter 3 - OMG IDL Syntax and Semantics*. 2001.

[36] Object Management Group. CORBA Portable Interceptors. Technical report, Object Management Group, 2004.

[37] Aleksey Gurtovoy. The Boost MPL Library. http://www.boost.org/libs/mpl/doc/index.html.

[38] Marjan Hericko, Matjaz B. Juric, Ivan Rozman, Simon Beloglavec, and Ales Zivkovic. Object Serialization Analysis and Comparison in Java and .NET. *SIGPLAN Not.*, 38(8):44–54, 2003.

[39] Chanika Hobatr and Brian A. Malloy. Using OCL-Queries for Debugging C++. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 839–840. IEEE Computer Society, 2001.

[40] BEA Systems Inc. Introduction to CORBA Request-Level Interceptors. `http://e-docs.bea.com/tuxedo/tux81/rli/rliover.htm`.

[41] SciTech Software Inc. (about) Open Watcom - Portable Compilers and Tools. `http://www.openwatcom.org/about/info_content.html`.

[42] SciTech Software Inc. Open Watcom - Portable Compilers and Tools. `http://www.openwatcom.org/index.html`.

[43] Yutaka Ishikawa. MpC++ Approach to Parallel Computing Environment. *SIGAPP Appl. Comput. Rev.*, 4(1):15–18, 1996.

[44] The JBoss Project. Javassist Home Page. `http://www.csg.is.titech.ac.jp/~chiba/javassist/`.

[45] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *Java(TM) Langauge Specification*. Addison-Wesley, 2000.

[46] Alan C. Kay. Smalltalk and Generic Concepts in Programming Languages. In *Proceedings of the International Conference on APL: Part 1*, page 340. ACM Press, 1979.

[47] Gregor Kiczales. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[48] Marc-Olivier Killijian, Juan-Carlos Ruiz, and Jean-Charles Fabre. Portable Serialization of CORBA Objects: A Reflective Approach. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 68–82. ACM Press, 2002.

[49] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The Case for Reflective Middleware. *Commun. ACM*, 45(6):33–38, 2002.

[50] Raimondas Lencevicius, Urs Holzle, and Ambuj K. Singh. Query-Based Debugging of Object-Oriented Programs. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 304–317. ACM Press, 1997.

[51] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification*. Addison-Wesley, 2nd edition edition, April 1999.

[52] Lisa Lippencott. Nitrogen. `http://nitric.sourceforge.net/`.

[53] Stanley B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, May 1996.

[54] Tom Lunney and Aidan McCaughey. Object Persistence in Java. In *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java*, pages 115–120. Computer Science Press, Inc., 2003.

[55] Sun Microsystems. Generics in the Java Programming Language. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[56] Sun Microsystems. Java 2 Platform, Standard Edition, v 1.4.2 API Specification. http://java.sun.com/j2se/1.4.2/docs/api/index.html.

[57] Sun Microsystems. Java Remote Method Invocation (Java RMI). http://java.sun.com/products/jdk/rmi/.

[58] Sun Microsystems. The Reflection API. http://java.sun.com/docs/books/tutorial/reflect/.

[59] Sun Microsystems. The JavaBeans Component Architecture. http://java.sun.com/products/javabeans/, 2003.

[60] The Eclipse Project. AspectJ Documentation. http://eclipse.org/aspectj.

[61] The GNU Project. DDD - Data Display Debugger. http://www.gnu.org/software/ddd/.

[62] The GNU Project. GCC Home Page - GNU Project - Free Software Foundation (FSF). http://gcc.gnu.org/.

[63] The GNU Project. GCJ: The GNU Compiler for Java. http://gcc.gnu.org/java/.

[64] The MICO Project. MICO - Mico Is COrba. http://www.mico.org.

[65] David Robson. Smalltalk. In *Proceedings of the 1983 Annual Conference on Computers : Extending the Human Resource*, page 133. ACM Press, 1983.

[66] Harald Rueß, Holger Pfeifer, and F.W. von Henke. Formalization and Reasoning in a Reflective Architecture. In M.H. Ibrahim, editor, *Reflection and Metalevel Architectures and their Applications in AI*, Montreal, Canada, 1995. IJCAI'95 Workshop.

[67] Sandeep S. Process Tracing Using Ptrace. *Linux Gazette*, (81), August 2002.

[68] Daniel Schulz and Frank Mueller. A Thread-Aware Debugger with an Open Interface. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 201–211. ACM Press, 2000.

[69] Brian C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, 1982.

[70] Brian Cantwell Smith. Reflection and Semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 23–35. ACM Press, 1984.

[71] Dennis M. Sosnoski. Java Programming Dynamics, Part 2: Introducing Reflection. http://www-106.ibm.com/developerworks/java/library/j-dyn0603/.

[72] The Squeak Project. Welcome to Squeak. http://www.squeak.org.

[73] Richard M. Stallman. *Using and Porting the GNU Compiler Collection (GCC)*. IUniverse.com, 2003.

[74] Kurt Stephens. XVF: C++ Introspection by Extensible Visitation. *SIGPLAN Not.*, 38(8):55–59, 2003.

[75] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[76] Norihisa Suzuki. Inferring Types in Smalltalk. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 187–199. ACM Press, 1981.

[77] The GCC Team. A Common C++ ABI for GNU/Linux. http://gcc.gnu.org/gcc-3.2/C++-abi.html.

[78] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 227–242. ACM Press, 1987.

[79] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002.

[80] Todd L. Veldhuizen. C++ Templates are Turing Complete. http://osl.iu.edu/~tveldhui/papers/2003/turing.pdf.

[81] Todd L. Veldhuizen. Template Metaprograms. http://osl.iu.edu/~tveldhui/papers/Template-Metaprograms/meta-art.html.