N3410 — Rich Pointers with Dynamic and Static Introspection

Dean Michael Berris (dberris@google.com)
Lawrence Crowl (crowl@google.com)
Matt Austern (austern@google.com)
Lally Singh (lally@google.com)

2012-09-18

Contents

1 Overview **2**

2 Problem Statement **3**

3 Proposed Solution **4**
3.1 Rich Pointers 4
3.2 Type Descriptors 5

4 Rationale **6**
4.1 Runtime Introspection 6
4.2 Fully Dynamic Extension 6
4.3 Runtime Upgrades and RPC 7

5 Implementation Hints **7**
5.1 Library-Only Approach 8
5.2 Library Interface + Compiler Assist 9
5.3 Compiler Assist 9
5.4 Metaprogramming 10
5.5 Inhibiting Type Descriptor Generation 10

6 Open Issues **10**

7 Related Documents **11**

8 References **11**

9 Acknowledgements **13**

A Appendix: Type Descriptor Details **13**

B Appendix: Static Metadata **14**

C Example: Generated Type Descriptors **16**

D Appendix: Using Static Metadata **18**

E Type Registration and Invalidation **19**

F Google Protocol Buffers Generic Serialization **21**

G Dynamic Plugins Example **22**

H JIT-Compiled RPC **23**

I Library Interface **24**

J Library Implementation: Annotations **25**

K Introspection in Programming Languages **26**
K.1 SmallTalk . . . 26
K.1.1 Compilation . . . 27
K.1.2 Syntax . . . 27
K.1.3 Type System . . . 28
K.1.4 Object Model . . . 28
K.1.5 Runtime . . . 28
K.2 Objective-C . . . 29
K.2.1 Compilation . . . 29
K.2.2 Syntax . . . 29
K.2.3 Type System . . . 30
K.2.4 Runtime . . . 32
K.2.5 Case Study: Distributed Objects . . . 33
K.2.6 Case Study: Serialization . . . 33
K.3 Java . . . 34
K.3.1 Compilation . . . 34
K.3.2 Syntax . . . 35
K.3.3 Type System . . . 35
K.3.4 Objects . . . 36
K.3.5 Runtime . . . 37
K.3.6 Case Study: Serialization . . . 39
K.3.7 Case Study: Distributed Objects . . . 39
K.4 OpenC++ . . . 39
K.4.1 Compilation . . . 40
K.4.2 Introspective Abilities . . . 40
K.4.3 Case Study: Distributed Objects . . . 40
K.4.4 Case Study: Serialization . . . 41

1 Overview

This paper proposes the definition of a standardized rich pointer and type descriptor construct in C++ to allow for standardized and efficient runtime and compiletime introspection of types and objects. Here we propose the semantics for a rich pointer and examples of programming problems addressed by this construct. We also provide an appendix on current reflection and introspection facilities.

2 Problem Statement

There are some application areas today where rich runtime information about objects is crucial. These application areas include:

- **Dynamic Adaptation.** Highly available server applications that have to stay up as a long running process and enable for dynamic adaptation to changing requirements. Upgrades to these applications currently need to be closely coordinated and tightly controlled, usually requiring that the service be actually brought down when an upgrade is required. There are solutions that exist which involve dynamically loaded libraries but changing the design of already packaged types in the system usually require downtime to rebuild and relink the binary.
- **Embedded Dynamic Languages.** Applications that embed dynamic programming languages typically settle for either a static interface to which the embedded dynamic language runtime, or a pure data interface implementing a (static) protocol between the embedded environment and the host application. Usually the dynamic types that can be generated in these embedded virtual machines are only usable in the context of these virtual machines limiting the interactions by which these embedded types interact with the host application.
- **Distributed Computing.** Distributed computing systems written in C++ are largely tied to static interfaces and types because of the limitation of the programming language. We currently do not have a standard programmatic way of dynamically generating types and referring to objects of these types and streaming objects of these types from one system to another. Current state of the art relies on code generators, domain specific languages, and even communication frameworks to achieve remote procedure calling and sharing state across elements in a distributed system.
- **Data Structure Refinement.** Applications use machine learning and dynamic modeling of environments through continuous refinement of data structures in memory rely on the capability to treat code as data, or at least be able to inspect the state and relationship between types in a hierarchy of types. The current limitations of the language force the applications that deal with constantly changing structures and types is to model them in runtime as merely data, losing much of the power of the programming languages runtime facility for efficiently modeling types and objects in the process.
- **Dynamic Characterization.** Graphical user interfaces typically rely on being able to inspect in-memory structures to represent graphical elements to be rendered on screen. Almost all the sufficiently advanced graphical user interface toolkits now use a staticallydefined and very rigid type system and implement a runtime metatype system because the programming language currently does not have a facility of doing proper rich runtime introspection of types.

There are several problems that this paper aims to address. Here are some of these problems:

- *How do we find out what the type of an object is at a given memory location?* Type erasure allows us to expose a generic API that works well across module boundaries, but being able to preserve type information across these module boundaries without having to rebuild and relink binaries is also as powerful.
- *How do we print the structure of a dynamic type at runtime?* Currently there are no ways to do this dynamically without resorting to manual, static, and expensive checks on types that are members of a statically-known hierarchy of types. There exists no standard means of knowing the type of a given object in memory at runtime with the current features of the language, especially when referred to using either a `void *` or a base type pointer.

- *How do we determine the relationship between any two given types at runtime?* The current way of doing this requires manual checks for whether one type can be dynamically cast to another, and inferring from the result what the potential relationship between the types are. There is currently no way of determining what types a given type is derived from, what type of inheritance (public, private, protected, virtual, etc.), and whether a given type is abstract or final, etc.
- *If we were able to create new types at runtime, how do we describe these types and inspect them?* For applications that rely on live updates for high availability and remote procedure calling systems, being able to reconstitute a type dynamically based on data obtained externally via I/O is crucial. Currently the only way to allow this is to implement a runtime type system by hand and perform all type inspections by hand, unable to leverage the rich type system that C++ provides. This is also important in applications where an embedded just-in-time (JIT) compiler can create new structures programmatically as well as allowing types generated in embedded runtime environments to be exported to the host application.

3 Proposed Solution

The problems raised above point to general dynamic programming utilities called runtime introspection and reflection. Enabling introspection and reflection at runtime has traditionally been costly and almost always requires the concept of a virtual machine for it to be doable. In this section we describe a mechanism for enabling introspection and potentially reflection without the need for a heavyweight virtual machine or runtime to make it possible.

There are two parts to this proposed solution: a smart pointer library and some compiler-assistance to generate static information. The smart pointer library provides the runtime introspection facilities, while compiler-assistance and static introspection capabilities along with additional traits cover the compile-time introspection facilities.

3.1 Rich Pointers

The smart pointer idiom has been generally accepted in practice largely for the utilities afforded to us by these smart pointers. The standard library now contains three smart pointers: `unique_ptr`, `shared_ptr`, and `weak_ptr`. These smart pointers encapsulate generic patterns for dealing with memory ownership and management and makes it transparent to users.

Following the smart pointer idiom this paper proposes a concept called rich pointers which not only carry the memory location of a given object but also a reference to an immutable representation of the type of this object (we call these *type descriptors*). An example usage of the `rich_ptr` is shown here:

```
struct foo {
    foo() {}
    ~foo() {}
};

rich_ptr<foo> p{new foo};
```

For the most part a rich pointer behaves like a normal pointer. Dereferencing will yield a reference to the object pointed to. A `rich_ptr` doesn't imply ownership semantics and is only meant to associate the type descriptor for a given normal pointer.

There are two primary operations supported on `rich_ptr` instances: getting the type descriptor and performing a `rich_cast`. The following example illustrates the semantics of both these operations.

```
rich_ptr<void> q = p;
rich_ptr<foo> r = rich_cast<foo>(q);
assert(type_descriptor(q) == type_descriptor(r));
```

Because p and q point to the same object in memory, getting the type descriptor of both rich pointers will yield the same type descriptor. In fact, any object of type `foo` in memory when referred to via a rich pointer will have the same type descriptor.

3.2 Type Descriptors

The metadata about an object's structure can only be generated by the compiler and therefore compiler assistance is required to support automatically generated type descriptors. These type descriptors can be inspected statically and dynamically. The runtime equivalent of the type descriptors are maintained by the language runtime and are tied directly to the runtime type information (RTTI) implementation.

To access a type descriptor at runtime, were introducing a standard function called `type_descriptor` which returns a pointer to an immutable object representing that unique type. Please refer to Appendix A for the complete details of the structure of this descriptor. To access a type descriptor at compile-time we're introducing a trait called `metadata`. Please refer to Appendix B for the complete details of the structure of this trait and other related traits.

What type descriptors contain is rich type information regarding a specific type. The idea is, all the types ever referred to via a rich pointer in a translation unit will expose a type descriptor. This type descriptor is then considered immutable and attempts to programmatically change an existing type descriptor invokes undefined behavior.

A compiler will be able to generate these type descriptors and make them available to the runtime implementation. Please refer to Appendix C for details of the example type descriptors generated for a simple case and Appendix D for the static descriptors examples.

There are two ways of making the runtime manage descriptors: registration and invalidation.

When a type descriptor is registered in the runtime, it must not allow the redefinition of an existing descriptor tied to a specific type. Attempts to register a different descriptor to an already registered type should fail. This enables dynamically loaded shared libraries to add new type descriptors to an existing runtime environment.

Invalidating a type descriptor does two things: invalidates all rich pointers to objects of the invalidated type – attempts to dereference the pointers of an invalidated type will then lead to exceptions that can be handled either by user code or by a special invalidation handler associated with the type. Invalidation handlers can be registered using the same mechanism by which new types are registered. This enables dynamically loaded shared libraries to invalidate types that it intends to either explicitly stop supporting or upgrade to a newer version.

Please refer to Appendix E for the proposed interface for making this happen manually either by explicit invalidation by module writers or implicitly by the compiler-generated module initialization routines.

4 Rationale

The current state of the art of dealing at runtime of types involves manipulating and traversing statically defined type descriptors. These type descriptors contain information about the types theyre supposed to represent (members, inheritance hierarchy, etc.) and are encapsulated with the same type they typically describe. This is useful for determining at runtime for example how to display certain pieces of information.

4.1 Runtime Introspection

An example of this use-case is the Qt frameworks' QMetaObject type that describes an object thats a member of the Qt type hierarchy. A QMetaObject represents things like type metadata (information associated with types as a key-value pair accessible at runtime) and methods (descriptions of the methods supported, accessed through an integer index).

The way types are annotated to provide this information is through the use of preprocessor macros to generate boilerplate code that registers this information as a static member of the type. Qt has a rich object system and a means for runtime dispatch of methods and traversal/inspection of types at runtime thats overlaid on top of the normal C++ mechanisms already available.

Another example of how richer runtime type data can be used to improve current implementations is how Google Protocol Buffers (ProtoBufs) use type descriptors to define how a protocol buffer is laid out in memory. This enables the protocol buffer messages to keep data in a serialized form in memory and support accessing this data in an efficient manner. The descriptor mechanism allows for parsing incoming data and determine whether the protocol buffer descriptor supports the data thats been received to construct an object of the correct type.

The down-side for both these approaches are:

- The information is represented as redundant information generated either mechanically (in the case of Google ProtoBufs using a proto compiler) or by hand (in the case of custom Qt types).
- Both mechanisms are very rigid and cannot trivially be updated dynamically. The static nature of the information generated does not lend itself to graceful upgrades while programs using these libraries are running.
- Most (if not all) of this rich information is largely only useful when working within the framework of these specific type systems. Mixing and matching ProtoBuf messages and Qt QObjects for instance so that information can be rendered in a GUI or streamed/saved to file typically require glue code to take care of the manual conversion from one type system to another.

See an example of how Google ProtoBufs can be implemented to not rely on ad-hoc reflection generated by the protobuf compiler and just leverage rich pointers and type descriptors in Appendix F.

We also provide a brief survey of introspection mechanisms from other programming languages and environments in Appendix K.

4.2 Fully Dynamic Extension

For network server applications there have been two major ways of achieving dynamic upgrades of a running system. The way the Apache Web Server does it as an example is to define a very strict protocol for how dynamic shared objects are described and behave – all extensions are implemented through this

system which also relies on the dynamic linker to resolve symbols when new modules are loaded into a running system. The other way is done by many games and dynamic systems is to embed a dynamic language runtime (like Lua) so that the parts that are meant to be upgraded at runtime can be implemented in a fully dynamic programming environment that supports this functionality natively without having to bring the whole application down.

Both these approaches are valid but they have their own trade-offs.

The dynamic module approach relies heavily on external dynamic linking solutions that have traditionally only been implemented in C and support only a C application binary interface (ABI). Although it is possible to hide the C++ implementation behind a C interface it limits the integration possibilities between host applications and dynamic modules. Furthermore this facility is largely platform dependent and is not standardized across vendor implementations.

Embedding a dynamic language runtime has the same limitations as typically the interaction between the host application and the embedded runtime are limited. The types generated or used in the context of the runtime typically have to be converted programmatically into something that the host application explicitly understands. The performance and maintenance costs of systems that do use embedded runtimes are also non-trivial.

In Appendix G we show an approach to adding dynamically registered types as plug-ins to an application.

4.3 Runtime Upgrades and RPC

In the realm of distributed systems programming the remote procedure call (RPC) pattern is very popular and many implementations rely on a mix of static types and dynamic structures to represent types. CORBA services rely on a very rigid client and server interface described using a domain specific language to define the interface by which clients can interact with the server. There are also a number of solutions for web services like SOAP (using XML for encoding RPC calls) and recently more popular REST interfaces (leveraging HTTP semantics and usually passing data encoded in JavaScript Object Notation (JSON)).

CORBA and SOAP services are typically mechanically generated using compilers that generate stubs later filled in by developers. These however suffer from the API versioning problem largely because there"s no way for the server to communicate with the clients the existence of new types and methods. Instead the coupling between the client and the server APIs require upgrades to be synchronized or in some cases have costly deprecation paths for client APIs that require the service to keep supporting an old version and the new version at the same time.

For web services written in C++ that use JSON as the data interchange format between client code (typically HTML pages with JavaScript running on web browsers) and server code, the only way to treat the data coming from the client is to treat it as a stream of bytes that are parsed and maybe generate statically typed objects to manipulate in the server. The other problem has to do with streaming hand-rolled or dynamic types as JSON or another data interchange format like XML.

See a hypothetical example of a JIT-compiling runtime-upgrading service that relies on dynamically generated types that are inspected and again upgraded using rich pointers in Appendix H.

5 Implementation Hints

This section provides information about how we could implement rich pointers in the context of C++11. We approach it by first writing a library solution that approximates the functionality using available

language features in C++11. We then show what the limitations of the language and runtime are along the way and work towards removing the limitations.

Here we present a first crack at providing rich pointer support using a template we"ll call `rich_ptr`. The goal of `rich_ptr` is to provide functionality for the definition and registration of types and objects of these types in the same object. Instances of `rich_ptr` will follow pointer semantics similar to what `shared_ptr` provides without the reference counting capability.

In Appendix I we show what the interface for a `rich_ptr` would look like. The minimal interface is by design meant to mimic how pointers work. It provides overloads to the dereference operator and the member access operator, and a member function `rich_ptr::get()` which returns the bare underlying pointer.

We then supply a means to programmatically defining type descriptors following the already presented type descriptors by hand in an attempt at a library-only approach.

5.1 Library-Only Approach

Is there a way to implement support for rich pointers using just C++11 features? We explore how this would look like in Appendix J which shows how this could be done with macros. Immediately there are a few limitations which we can identify:

- Without compiler assistance, users will have to annotate types that they want to generate metadata for with preprocessor macros. The syntax of the macros typically deviates too much from the normal C++ syntax for class definitions.
- Because you have to consciously annotate the type youre registering, this makes it hard to manage especially if the types definition is not something you can change (i.e. when using a third-party provided library).
- This will not properly handle dynamically loaded libraries at runtime and will be at the mercy of the implementation and platform. Because preprocessor macros are typically only valid very early on in the compilation process, the annotations are no longer available in the translation unit.

With a library-only implementation we also cannot do the following reliably:

- **Safe runtime type invalidation.** This will require two things:
  - A standardized registry of types available and managed at runtime.
  - Well defined module initialization/cleanup semantics and interfaces.
- **Seamless invalidation at runtime.** Because of the linkage and concurrency issues introduced by standard-allowed optimizations, it is not guaranteed that library-defined invalidation routines can be protected. The worst case scenario is that code paths and branches that can be optimized appropriately using normal pointers as opposed to rich pointers are in jeopardy of behaving much poorly.
- **Safe runtime type creation.** For JIT compilers at the moment, only code and data thats accessible through opaque pointers (`void *`) and rigidly-defined function pointers that have primitive return types can be integrated into the hosting runtime context. New types cannot be safely registered and invalidated without explicit runtime support of type registration.
- **Automatic descriptor generation.** With the library approach types need to be explicitly registered and annotated. The compiler on the other hand has all the information it needs and can

generate the type information at compilation time and can do so only for types that are used in rich pointers and types pointed to by rich pointers.

A previous version of this paper (N3340) was proposing to add a new pointer type. The downside to that approach would be that the work required to change the standard defining a new pointer type is not enough to justify the gains.

This version of the paper introduces a hybrid approach which shows what rich pointers would look like with a library interface (`rich_ptr`) while introducing some compiler assistance functionality (`type_descriptor` and `metadata`).

5.2 Library Interface + Compiler Assist

The library interface reduces the reliance on changes to the C++ language and allows alternate implementations. There are a number of parts to the library implementation that are implementation-defined:

- Runtime registration/lookup of type descriptors.
- Invalidation of type descriptors.
- The actual storage mechanism for the handle to a type descriptor for each instance of a `rich_ptr`.
- How type-lists are implemented in the case of `metadata` and associated types.

Please refer to Appendix I for all the details of the proposed `rich_ptr` type.

The reason for implementation-defined runtime registration/lookup implementations is a consequence of the reality where different platforms provide different implementations of dynamic linkage. Without a standardized facility for defining shared libraries or shared objects there's no choice but to leave out these specifications to be defined by the implementation. The same reasoning applies to invalidation of type descriptors.

One approach to the mapping between a pointer and a type descriptor in the face of invalidation at runtime is to have a token representing an intermediate mapping. This allows an implementation to provide two-stage lookup and to maintain a single definition of valid type descriptors in a globally accessible map.

An implementation is also allowed to tag objects in memory (similar to how RTTI is done) to encode the token to the object. Some compilers already encode debugging information as part of the translation unit – this metadata can be used at runtime as well in debug builds. Other implementation approaches for runtime introspection are possible and the paper tries not to preclude potentially valid implementations.

5.3 Compiler Assist

There are two main parts where compiler assistance comes in: automatic generation of type descriptors, and static metadata. The `metadata` metafunction yields a compiler-generated type that acts as a handle to a type's definition. This same metadata is meant to be consistent with the runtime type descriptors.

The handle is implementation-defined and is not required to be a valid C++ type. A whole family of traits associated to the handle type are also defined.

5.4 Metaprogramming

This paper relies on the current state of the art of using template metaprogramming to provide the interface for the traits that can be applied to the metadata handle. Unfortunately this interface is severely limited. One potential way to alleviate the need for template metaprogramming is to extend `constexpr` functions to support pure `constexpr` functions.

With pure `constexpr` functions:

- We can define functions which have absolutely no runtime side-effects.
- We can use imperative control structures that operate at compile time.
- Implement generative type functions.
- Treat literals as first class objects when performing type-system checks (e.g. compare string equivalence per translation unit).

Although pure `constexpr` functions will make metaprogramming easier, it is not a necessity in providing the rich pointer functionality.

5.5 Inhibiting Type Descriptor Generation

An the Kona meeting questions were raised about whether it should be possible to inhibit type descriptor generation. There are a number of approaches that can be taken for type descriptor generation:

- **Default inhibit, annotate for generation (opt-in generation).** This is the conservative approach. This makes it really hard to get the utility to be useful to as many people as possible when implementations get around to supporting the proposed facilities.
- **Default generate, annotate for exclusion (opt-out generation).** This is the aggressive approach which would turn it on for all types except for certain types. The suggestion is to have this be controlled by type annotations which should be supported by implementations.
- **Implementation defined.** This could be made optional similar to how some implementations actually enable/disable RTTI or exception support.

Of the three above the preferred solution is to define an opt-out mechanism using type annotations.

6 Open Issues

At the time of this writing there are two open issues with the runtime type system registration and invalidation. We enumerate these below which require committee feedback and further investigation for future revisions to the paper:

- **Dynamic invalidation of statically registered types.** Currently it is implied that the registration mechanism for statically-linked type descriptors is implementation-defined, and that the registry mechanism may vary. Future revisions to the paper should have this addressed based on feedback from the committee.
- **Type name to descriptor mapping.** As C++ does not define link-time requirements on type names and how they map to actual objects from different modules, introducing a mapping based on a type's name may be problematic. What comes to mind are types with the same name in the

anonymous namespace in two different translation units. Whether there's a name conflict is unclear, and an implementation may have to disambiguate the type's names using some other mechanism than just the mangled symbol name.

- **Shared library semantics are platform-dependent and undefined by the standard.** A lot of the invalidation and shared-library/module specific information in the paper assumes a consistent behavior of dynamically loaded libraries. Without clear standards on how shared-libraries work across platforms much of the guarantees may not be made on all current platforms. Feedback on the direction for shared/dynamic library semantics and whether these should be standardized would be greatly appreciated.

It is the authors' hope that these issues would be at least discussed if interest in getting runtime introspection into the standard at some point in the future.

7 Related Documents

N3340 - Rich Pointers by Dean Michael Berris, Matt Austern, and Lawrence Crowl.

N2316 - C++ Modules (revision 5) by Daveed Vandevoorde

In the C++ Modules proposal, the initialization order and means for introducing a way for formally supporting dynamic libraries (as introduced by **N1400 - Toward Standardization of Dynamic Libraries** by Matt Austern).

N1976 - Dynamic Shared Objects Survey and Issues

This paper largely goes through the various papers regarding the topic of shared objects and dynamic libraries.

N1751 - Aspects of Reflection in C++ by Detleff Vollmann **N1775 - A Case for Reflection** by Walter Brown et al

These two papers describe generally how Reflection and Introspection would be approached, but lacked specific implementation details that this paper partly provides.

8 References

[1] 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.

[2] 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.

[3] Shigeru Chiba. OpenC++ Home Page. `http://www.csg.is.titech.ac.jp/~chiba/openC++.html`.

[4] Shigeru Chiba. OpenC++ Tutorial. `http://www.csg.is.titech.ac.jp/~chiba/opencxx/tutorial.pdf`.

[5] Shigeru Chiba. A Metaobject Protocol for C++. In *Conference on Object Oriented Programming Systems Languages and Applications*. SIGPLAN: ACM, ACM, 1995.

[6] Apple Computer. Cocoa: The Objective-C Programming Language. `http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/index.html`.

[7] Apple Computer. Cocoa. `http://developer.apple.com/cocoa/`, 2003.

[8] Steven Dekorte. Objective-C. `http://www.dekorte.com/Objective-C/index.html`.

[9] 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.

[10] The GNUstep Project. GNUstep.org. `http://www.gnustep.org/`.

[11] 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.

[12] 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.

[13] 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.

[14] Yutaka Ishikawa. MpC++ Approach to Parallel Computing Environment. *SIGAPP Appl. Comput. Rev.*, 4(1):15–18, 1996.

[15] The JBoss Project. Javassist Home Page. `http://www.csg.is.titech.ac.jp/~chiba/javassist/`.

[16] 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.

[17] 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.

[18] Sun Microsystems. Generics in the Java Programming Language. `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`.

[19] 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`.

[20] Sun Microsystems. Java Remote Method Invocation (Java RMI). `http://java.sun.com/products/jdk/rmi/`.

[21] Sun Microsystems. The JavaBeans Component Architecture. `http://java.sun.com/products/javabeans/`, 2003.

[22] The GNU Project. GCJ: The GNU Compiler for Java. `http://gcc.gnu.org/java/`.

[23] The Squeak Project. Welcome to Squeak. `http://www.squeak.org`.

[24] 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.

9 Acknowledgements

Thanks to the following people who have contributed their feedback on the N3340 draft and the overall concept: David Abrahams, Roman Perepelitsa, Jason Bolla, Lawrence Crowl, and Matt Austern.

This revision (N3410) has the contributions of Lally Singh and Lawrence Crowl. Questions and feedback raised by Jens Maurer, Alisdair Meredith, Ville Voutilainen, David Vandevoorde, Walter Brown, and Faisal Vali have influenced the revisions to the original (N3340) paper.

A Appendix: Type Descriptor Details

The following structure definitions show the details that type descriptors can contain. Here we describe a means for exposing descriptors for functions (`function_descriptor_t`), member data (`field_descriptor_t`), and classes/structs (`type_descriptor_t`).

```
struct function_descriptor_t {
  char const *name;  // null terminated function name
  type_descriptor_t const *result_type;
  type_descriptor_t const *args[];
  enum { NORMAL, VIRTUAL, CONST, VOLATILE,
         CONSTEXPR, STATIC, NAMESPACE } qualifiers_t;
  int qualifiers;  // binary ORed qualifiers_t values.
  enum { CONSTRUCTOR, DESTRUCTOR, FREE, MEMBER }
  function_type_t;
  function_type_t function_type;
  type_descriptor_t *member_of;  // pointer to enclosing type.
  unspecified_type callable;
};
```

```
struct field_descriptor_t {
  char const *name;  // null terminated member name
  type_descriptor_t const *type;
  enum class access_qualifiers_t { PRIVATE, PUBLIC, PROTECTED };
  access_qualifiers_t access_qualifier;
  enum class membership_qualifiers_t { INSTANCE, STATIC };
  membership_qualifiers_t membership_qualifier;
};

struct type_descriptor_t {
  char const * name;  // null terminated type name

  struct inheritance_descriptor_t {
    enum class access_type_t { PRIVATE, PUBLIC, PROTECTED };
    access_type_t access_type;
    enum class inheritance_type_t { FINAL, VIRTUAL };
    inheritance_type_t inheritance_type;
    type_descriptor_t const *base;
  };
```

```
    inheritance_descriptor_t *bases[];

    struct method_descriptor_t {
      enum class access_type_t { PRIVATE, PUBLIC, PROTECTED };
      access_type_t access_type;
      enum class inheritance_type_t { NORMAL, FINAL, VIRTUAL };
      inheritance_type_t inheritance_type;
      function_descriptor_t const *type;
    };

    method_descriptor_t *methods[];
    field_descriptor_t *members[];
    size_t size;
  };
```

The prototypes for the `type_descriptor` function are given below:

```
  template <class T>
  type_descriptor_t const *
  type_descriptor(rich_ptr<T>);

  template <class T>
  type_descriptor_t const *
  type_descriptor(T*);

  template <class R, class T...>
  function_descriptor_t const *
  type_descriptor(R(T...) *);  // for function pointers.
```

B Appendix: Static Metadata

In this appendix we describe the traits and types associated with the proposed `metadata` trait.

```
  template <class T>
  struct metadata {
    typedef unspecified_type type;
  };
```

As an analog to the `type_descriptor_t` defined in Appendix A we present the following suggested traits for inspecting the special type `metadata<T>::type`.

```
  template <class T>
  struct fields {
    typedef unspecified_type type;
  };
```

The `fields` metafunction is meant to apply to the `metadata` associated with a given type. It yields a list of types which can then each be queried with the following traits:

```
  template <class T>
  struct name {
```

```
  static constexpr const char* value = unspecified;
};

template <class T>
struct access_qualifier {
  enum class access_qualifiers_t { PRIVATE, PUBLIC, PROTECTED };
  static constexpr access_qualifiers_t value = unspecified;
};

template <class T>
struct membership_qualifier {
  enum class membership_qualifiers_t { INSTANCE, STATIC };
  static constexpr membership_qualifiers_t value = unspecified;
};
```

Another metadata trait would be to get the functions associated with a given type (if there are any). We propose a special trait called `functions` for this purpose:

```
template <class T>
struct functions {
  typedef unspecified_type type;
};
```

Just like the `fields` trait, this applies to a `metadata\<T\>::type`. It yields a list of types which can then each be queried with the traits that apply with the result of `fields` and the following additional traits:

```
template <class T>
struct result_type {
  typedef unspecified_type type;
};

template <class T>
struct args {
  typedef unspecified_type type;
};

template <class T>
struct qualifiers {
  static constexpr int value = unspecified;
};

template <class T>
struct function_type {
  enum class function_type_t { CONSTRUCTOR, DESTRUCTOR, FREE, MEMBER };
  static constexpr function_type_t value = unspecified;
};

template <class T>
struct member_of {
  typedef unspecified_type type;
};
```

The `args` trait will yield a type-list similar to what `metadata` will provide.

The `qualifiers<T>::value` value will be similar to what `function_descriptor_t::qualifiers` contains.

The `member_of` trait will yield a type similar to what `metadata` will provide.

In all these traits, the implementation decides what the types and values are (or whether they are available on inhibited for certain types).

C Example: Generated Type Descriptors

In this section we take an example type and walk through an example generated code that compilers are meant to generate for type descriptors. Consider the following example case:

```
#include <iostreams>
#include <string>
#include <runtime>

struct foo {
  foo() : a_(0), b_() {}
  foo(foo const &) = delete;
  foo(foo &&) = delete;
  foo& operator=(foo) = delete;
  void bar() { /* do nothing */ };
  ~foo() { /* do nothing */ };
 protected:
  int a_;
 private:
  std::string b_;
};

int main(int argc, char *argv[]) {
  rich_ptr<foo> f{new foo};
  // The following should print ::foo.
  std::cout << type_descriptor(f)->name << std::endl;
  delete f.get();
  return 0;
}
```

In the above listing we rely on the `type_descriptor` of the rich pointer f that mimics the functionality of `std::type_info`. What the compiler generates then relates to module loading at run-time (in this case is run before `main` is called) and the associated type information that users will be able to access through the `type_descriptor_t` object resultant to the call to `type_descriptor`:

```
// Type descriptors, generated by compiler and registered through the
// runtime interface. Note that only types that are used through the
// rich pointer syntax will automatically have the type descriptors
// generated for them by the compiler along with the dependent types.
void __register_types() {
  // For convenience...
```

```
typedef type_descriptor_t::method_descriptor_t method_descriptor_t;
static function_descriptor_t const foo_void_bar {
  ::foo::bar,  // function name.
  &void_type,  // result type, primitive types defined in <runtime>.
  {nullptr},  // no arguments.
  function_descriptor_t::NORMAL,  // qualifiers.
  function_descriptor_t::MEMBER,  // type.
  nullptr,  // member_of, nullptr at this time.
  std::bind(&foo::bar, _1),  // callable
  function, 1st arg is this.
};

static function_descriptor_t const foo_ctor_0 {
  ::foo::foo,  // function name.
  nullptr,  // result type, nullptr for constructors.
  {nullptr},  // no arguments.
  function_descriptor_t::NORMAL,  // qualifiers.
  function_descriptor_t::CONSTRUCTOR, // type.
  nullptr,  // member_of, nullptr at this time.
  &__constructor<foo>::callable,  // implementation defined.
};

static function_descriptor_t const foo_dtor {
  ::foo::~foo,  // function name.
  nullptr,  // result type, nullptr for destructors.
  {nullptr},  // no arguments.
  function_descriptor_t::NORMAL,  // qualifiers.
  function_descriptor_t::DESTRUCTOR,  // type.
  nullptr,  // member_of, nullptr at this time.
  &__destructor<foo>::callable,  // implementation defined.
};

static type_descriptor_t const foo_type {
  ::foo,  // type name.
  {nullptr},  // no bases.
  {
    {
      method_descriptor_t::PUBLIC,
      method_descriptor_t::NORMAL,
      &foo_ctor_0
    },
    {
      method_descriptor_t::PUBLIC,
      method_descriptor_t::NORMAL,
      &foo_void_bar
    },
    {
      method_descriptor_t::PUBLIC,
      method_descriptor_t::NORMAL,
      &foo_dtor
```

```
      },
      nullptr
    },  // functions.
    {
      {
        a_,  // member name.
        &int_type,  // member type, primitives defined in <runtime>.
        field_descriptor_t::PROTECTED,  // access qualifier.
        field_descriptor_t::INSTANCE  // membership qualifer.
      },
      {
        b_,  // member name.
        &string_type,  // member type, in compilation of <string>.
        field_descriptor_t::PRIVATE,  // access qualifier.
        field_descriptor_t::INSTANCE  // membership qualifier.
      }
    },  // members.
    sizeof(foo)  // the statically determined size.
  };

  // We then wire up the members.
  const_cast<function_descriptor_t*>(&foo_ctor_0)->member_of = &foo_type;
  const_cast<function_descriptor_t*>(&foo_void_bar)->member_of = &foo_type;
  const_cast<function_descriptor_t*>(&foo_dtor)->member_of = &foo_type;

  // This part is the implementation defined part.
  std::__runtime_register_type_descriptor<foo>(&foo_type);
}
```

It is intended that the hidden generated function `__register_types` will be invoked at module initialization time. Depending on whether dynamically loaded modules will allow user-defined module intialization/cleanup interfaces so that users can manually register types and explicitly invalidate types already registered, the intent is that the compiler-generated descriptors will still get registered at module scope (internal linkage) and would have to be explicitly registered at global scope in order to be exported at runtime. It may be sufficient to use implementation-defined means of marking types as exported as part of the library interface for shared modules but the intent is to define a standardized API for this mechanism that works across platforms.

D Appendix: Using Static Metadata

In this section we show how we can use the static metadata. We start with a minimal example similar to Appendix C where we print the name of the type.

```
#include <iostream>
#include <metadata>  // proposed header.

struct foo {};

int main(int argc, char* argv[]) {
```

```
  using namespace std;
  // The following should print "::foo"
  cout << name<metadata<foo>>::value << endl;
  return 0;
}
```

Because we can make inquiries on the types enabled by `metadata` we can use this in SFINAE as well as in `static\_assert` situations:

```
template <class T>
constexpr void range_concept_check(T) {
  static_assert(name<metadata<T>>::value != nullptr, "We need static reflection!");

  struct begin { static constexpr const char* value = "begin"; };
  struct end { static constexpr const char* value = "end"; };

  // We look for a function named begin and end that return the same types.
  // In this part we hand-wave away some metafunctions.
  static_assert(
    is_same<
      result_type<
        find<
          functions<metadata<T>>,
          equals<
            name<_>::value,
            begin::value
          >
        >
      >,
      result_type<
        find<
          functions<metadata<T>>,
          equals<
            name<_>::value,
            end::value
          >
        >
      >
    >::value, "To be a range, begin and end should return the same type.");
}
```

E Type Registration and Invalidation

There are two functions that are being proposed as standard extensions in the `<runtime>` header: `register_type` and `invalidate_type`.

`register_type` is meant to be called at module initialization time depending on how dynamic/shared or static modules are initialized. The result is a boolean indicating success and a pointer to the new immutable type descriptor. This will only return true if the type T is valid and that it has not been

previously registered.

```
// Example usage:
//     bool ok;
//     type_descriptor_t const *p;
//     tie(ok, p) = register_type(make_rich<foo>());
//     assert(ok);
template <class T>
tuple<bool, type_descriptor_t const *>
register_type(rich_ptr<T>);
```

`invalidate_type` is meant to be called at module initialization time depending on how dynamic/shared or static modules are initialized. The result contains a boolean indicating success and pointers to the invalidated type descriptor and new type descriptor. It is important that this is called at module initialization time when the current runtime context and module-specific runtime contexts are available. Calls to `new` within module scope will always use the module-specific runtime context. During module initialization, calls to `type_descriptor(...)` will always refer to the current runtime context. If a type is not already registered through `register_type(...)` then calling `invalidate_type` will return the tuple (`false`, `nullptr`, descriptor to new type).

```
template <class T>
tuple<bool, type_descriptor_t const *, type_descriptor_t const *>
invalidate_type(type_descriptor_t const *, rich_ptr<T>,
                function<bool(type_descriptor_t const *, rich_ptr<T>&)>);
```

The following listings show example example usage of the `invalidate_type` function.

```
bool ok;
type_descriptor_t const *old_type, *new_type;
rich_ptr<foo> old{nullptr};
tie(ok, old_type, new_type) =
    invalidate_type(type_descriptor(old), make_rich<foo>());
assert(new_type != old_type && ok);
```

Another case is for "deregistering" a type, by simply providing nullptr to the second argument to `invalidate_type`:

```
tie(ok, old_type, new_type) =
    invalidate_type(type_descriptor(old), nullptr);
assert(new_type == nullptr && type_invalidated(old) && !ok);
```

The above is suggested for module cleanup time when types only meant for module scope should be deregistered and all rich pointers for these types should be invalidated.

The third argument to `invalidate_type` is the invalidation handler function. This function is called when a rich pointer of an invalidated type is dereferenced. It is passed the old type descriptor and a reference to the invalidated rich pointer. The invalidation handler should return true if it should allow the application to continue execution and false to force a call to `std::abort`. The default handler throws an `invalidated_ptr` exception which contains the invalidated pointer and a pointer to the old type descriptor.

A convenience function for explicitly checking whether the type of a given pointer has been invalidated.

```
bool type_invalidated(rich_ptr<void>);
```

There are only ever at most two versions of a type descriptor for any given registered type.

F Google Protocol Buffers Generic Serialization

An example of how a Google Protocol Buffer (protobuf) serialization mechanism would work is shown below as a function that takes any protobuf object and serializes it to an output stream.

```
// We want to ensure that the pointer were going to take does
// point to an object statically derived from proto::Message. For
// this we leverage the normal C++ static rules for inheritance
// and polymorphism.
namespace proto {
  bool serialize(rich_ptr<Message> m, std::ostream &os) {
    // In here we can then inspect the type associated with the pointer m
    // reflecting the runtime representation.
    type_descriptor_t const *type = type_descriptor(m),
                            *msg_type = type_descriptor<Message>(nullptr);

    // We then traverse the members of the protocol buffer and perform a generic
    // lookup of the data so we can encode properly into the output stream.
    std::string buffer;
    for (field_descriptor_t const *field : type->members) {
      if (field == nullptr)
        break;
      if (!m->get_member(field->name, &buffer))
        return false;
      os.write(s.data(), s.size());
    }
    return true;
  }
}
```

For exposition and completeness, we provide an example implementation of `proto::Messages` `get_member` function and enclosing scope using rich pointers and introspection functionality:

```
namespace proto {
  class Message {
    char *buffer_;  // dynamically initialized by derived classes
    static std::map<std::string, type_descriptor_t const *> registry;

   protected:
    explicit Message(size_t derived_size)
    : buffer_(new (std::nothrow) char[derived_size]) {}

   public:
    virtual bool get_member(std::string const &name, std::string *buffer) {
      type_descriptor_t const *type =
          registry[type_descriptor(rich_cast<Message>(this))->name];
      bool ok = false;
      size_t offset = 0;
      field_descriptor_t const * field = nullptr;
      tie(ok, offset, field) = get_offset(name, type->members);
      if (!ok) return false;
```

```
      size_t size = field->type->size;
      buffer->assign(buffer_ + offset, buffer_ + size);
      return ok;
    }

    virtual ~Message() {
      delete [] buffer_;
    }
  };
}
```

G Dynamic Plugins Example

Here we show one approach to adding dynamically registered types as plug-ins to a hypothetical server using rich pointers:

```
// We want to have a global registry of handler objects mapped to URLs.
static map<string, pair<rich_ptr<void>, function<void<string>>>> handlers;

// Then we define a way for a new module to register completely new
// objects at runtime:
bool register_handler(std::string const &url,
                      rich_ptr<void> handler,
                      bool replace) {
  // Here we can determine whether the object being registered as a
  // handler supports the required API (at runtime!)
  type_descriptor_t const *type = type_descriptor(handler);
  if (type == nullptr) return false;

  bool compliant = true;
  type_descriptor_t const *string_type =
      type_descriptor(rich_ptr<std::string>());

  function<void(string)> f;
  for (method_descriptor_t const *method : type->methods) {
    if (method == nullptr) break;
    compliant = compliant || (
        equals(method->type->name, get)
        && method->access_type == method_descriptor_t::PUBLIC
        && length(method->type->args) == 2
        && method->type->result_type == nullptr
        && method->type->args[1] == string_type);
    if (compliant && !f.get()) f = bind(method->type->callable, f, _1);
  }
  if (!compliant) return false;
  if (!replace && handlers.find(url) != handlers.end()) return false;
  handlers[url] = make_pair(handler, f);
  return true;
```

```
}

// When were ready to handle a URL, we do the following:
bool handle_url(std::string const &url, std::string const &request) {
  auto it = handlers.find(url);
  if (it != handlers.end()) {
    try {
      (*it)(request);
    } catch (...) {
      return false;
    }
    return true;
  }
  return false;
}
```

H JIT-Compiled RPC

In this section we provide a hypothetical service that relies on the notion of a JIT compiler hosted by an application that allows for dynamically generating types and objects that can be referred to using rich pointers.

The following function takes some arbitrary input and returns a function pointer to a compiled function that returns objects via a rich pointer and takes context through a rich pointer as well.

```
function<rich_ptr<void>(rich_ptr<void>)>
  Compile(string const &input, rich_ptr<void> data);
```

Given the example for the network server, lets consider an object that exposes its methods as RPC calls.

```
class HelloWorldRPC {
  string url_;
 public:
  explicit HelloWorldRPC(url const & url)
  : url_(url) {}

  void get(string const &input);
 private:
  void update(string const &input);
};

void HelloWorldRPC::get(string const &input) {
  rich_ptr<HelloWorldRPC> self = make_rich<HelloWorldRPC>(this);
  istringstream tokens(input);
  string function;
  tokens >> function;
  type_descriptor_t const *this_type = type_descriptor(self);
  for (type_descriptor_t::method_descriptor_t const *method :
    this_type->methods) {
    if (method == nullptr) {
```

```
      cerr << "Cannot find function " << function << '\n'.
      return;
    }
    if (function == method->type->name) {
      if (length(method->type->args) != 2) {
        cerr << "Cannot invoke function " << method->type->name
             << " for bad signature.\n"
      } else {
        // The following call will follow C++ member access rules.
        // In case the calling scope (determined by self) is not
        // the correct type or is not a friend scope, then the call will
        // throw ‘bad_function.
        method->type->callable(self, input);
        return;
      }
    }
  }
}

void HelloWorldRPC::update(string const &input) {
  // In here we then compile the input and register a new handler
  // that is yielded by the returned function.
  rich_ptr<HelloWorldRPC> self = make_rich<HelloWorldRPC>(this);
  auto factory = Compile(input, self);
  if (factory.get()) {
    // Replace the old handler with the new one!
    auto old = handlers.find(this->url_);
    if (register_handler(this->url_, factory(), old != handlers.end())) {
      if (old != handlers.end()) delete it->first;
    }
  } else {
    cerr << "Cannot update, Compile complains.\n";
  }
}
```

The proposed feature enables for easily maintaining runtime types and allowing for self-updating applications that host fully dynamic runtimes.

I Library Interface

```
// Usage:
//   std::rich_ptr<foo> p{ new foo };
template <class T>
struct rich_ptr {
  rich_ptr() = default;
  explicit rich_ptr(T *ptr);
  rich_ptr(rich_ptr const &other) = default;
  rich_ptr(rich_ptr &&other) = default;
```

```
  rich_ptr& operator=(rich_ptr);
  rich_ptr& operator=(T*);
  T* get() const;

  ~rich_ptr() = default;
  T& operator*() const;
  T& operator->() const;

  // We also want rich_ptr<T> to be convertible to T*.
  operator T*() const;
 private:
  T *ptr_;
  unspecified_type *descriptor_;
  friend template <class V> type_descriptor_t const *
    type_descriptor(rich_ptr<V> const &);
  friend template <class V> bool
    type_invalidated(rich_ptr<V> const &);
  friend template <class V> tuple<bool, type_descriptor_t const *>
    register_type();
};
```

J Library Implementation: Annotations

A few examples of the preprocessor-macro approach to annotating types for type descriptor generation is shown below.

```
template <class T>
tuple<bool, type_descriptor_t const *>
register_type() {
  static_assert(false, No explicit definition for type provided.);
  return make_tuple(false, nullptr);
}

// Lets annotate the type and wrap it in macros so that we can
// programmatically generate the explicit overload for the register_type
// template function.
DEFINE_REGISTERED_CLASS(foo)
  REGISTERED_CONSTRUCTOR(foo()) = default;
  REGISTERED_DESTRUCTOR(~foo);
  REGISTERED_MEMBER(void bar());
  foo(foo &&) = delete;
  foo(foo const &) = delete;
  foo& operator=(foo) = delete;
  PRIVATE_MEMBER(int a_);
  PRIVATE_MEMBER(string b_);
END_REGISTERED_CLASS_DEFINITION(foo);

// We provide explicit overloads and definitions for a given type.
```

```
// This is generated by the macros above right where the class is
// defined (in END_REGISTERED_CLASS_DEFINITION).
template <>
tuple<bool, type_descriptor_t const *>
register_type<foo>() {
  static type_descriptor_t const foo_type {
    /* Defined as shown earlier in the paper. */
  };
  // We then duplicate this definition in the rich_ptr<T> static
  // unique_descriptor.
  rich_ptr<T>::unique_descriptor = &foo_type;
  return make_tuple(true, &foo_type);
}

// We then call the explicit registrations for types we want to use in
// main(), or use static initialization of globals to rely on invoking
// the register_type<...>() function for all registered types.
int main(int argc, char *argv[]) {
  register_types<foo>();  // A variadic template function that takes
                          // a list of types to actually register at
                          // the start of main.
  // as main would normally be.
  return 0;
}
```

K 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 [21] 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 Objective-C 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++[5] provides a mechanism for compiler plug–ins to extend the language itself.

K.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.

K.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 [23].

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.

K.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`.

K.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[24]; 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` .

K.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[11], without the 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.

K.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.

[1] [2] extended this concept further into *explicit metaclasses*[9], resulting in the Classtalk platform
[2] Without typed variables, such a mechanism would not be useful anyways

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.

K.2 Objective-C

Objective–C[6] is a SmallTalk–derived object–oriented layer added to standard C. Originally developed by Brad Cox and the StepStone Corporation[8], 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[8]. 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.

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[10] and Apple's implementations use the new prefix.

K.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.

K.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.

K.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 `@implementation`. 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 `interfaces` 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.

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

K.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[7]). 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 `autoreleased` 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.

K.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

K.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.

K.3 Java

The Java 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[15].

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[22] that eliminate the need for a virtual machine and its overhead.

K.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.

K.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 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 1: Simple Java Program

K.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!

K.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[18]. 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.

K.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.

K.3.6 Case Study: Serialization

Java provides built–in serialization[17, 12, 1]. By implementing a zero–method interface `Serializable`, an object can be serialized. The primitive types can also be serialized.

Use `ObjectOutputStream`[19], 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.

K.3.7 Case Study: Distributed Objects

Java provides a basic distributed objects mechanism called Remote Method Invocation (RMI)[20]. 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.

K.4 OpenC++

OpenC++[5, 3, 4] 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[16, 13]. 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[14] has been used specifically for the high–performance computing arena.

K.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`.

K.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`.

K.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.

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

Figure 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.

K.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 3 has members of each type.

```
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 3: OpenC++ Class with Serialization Keywords

The metaclass would have to keep track of which `DataObjects` 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`).