Rochester Institute of Technology
Computer Science Department

Dynamic Encapsulation of C++ Objects

by
Frank E. Barrus

A thesis, submitted to
The Faculty of the Computer Science Department
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

____
Professor Andrew T. Kitchen

____
Professor Warren R. Carithers

____
Professor Peter G. Anderson

February 12, 2000

Acknowledgments

I would like to thank the following people for helping out with this thesis, directly or indirectly, in no particular order:

— Andrew Kitchen, Warren Carithers, and Peter Anderson for being my thesis advisors and being patient for many years while I strove to complete my goals.

— Tad, for many brainstorming sessions and document reviews.

— Various people on the Internet (you know who you are) who tried out DECO and SHACC and provided feedback.

— The many past members of R.I.T.'s Computer Science House, for providing an environment that fostered a creative technical education, and the current members of C.S.H. for maintaining the systems upon which my thesis web page is hosted.

— Karl and Marty, my managers at Xerox, for giving me some time on Fridays for my thesis work, and Denise, Steve, Jean, Aaron, Ray, John, Bill, Chris, Kevin, and Lou for providing a mixture of support and chiding. (also some much needed escape and exercise: rollerblading along the canal!)

— The Management of the now-defunct Inferno Operating Systems venture at Murray Hill (Bell Labs, Lucent Technologies) for agreeing to give me six weeks off for my thesis work. (if only the organization hadn't been shut down right after I got the time off approved!) Also Chris, Mike, Jimbo, Tom, Xiang, Ravi, Eric, Tad, Bill, Bruce, Dave, Craig, Larry, Clay, John, Anthony, Don, Sam, Matt, Ed, Atul, Amy, Diana, Tara, Michele, and everyone else from Inferno for their support and their attempts to push me to get this done.

— Larry, Matt, and Craig, for being accepting of me taking a few weekends off from our new startup (savaJe technologies) to get this paper finished.

— Many past and present researchers and developers who provided ideas and inspiration (see bibliography for details).

— My parents, family, and friends, for waiting patiently for the official completion of my degree.

— Dawn, for brainstorming, reviews, and being understanding and supportive through some of the busiest and most stressful days of my life. They're finally over!

ABSTRACT

Classes in C++ provide static encapsulation of objects, by generating code which contains specific knowledge about the internals of encapsulated objects. Static encapsulation occurs at compile time, and therefore cannot directly support the evolution of objects, since recompilation of source code is required if the internal layout changes. This also prohibits the use of distributed or persistent objects without first ensuring that the internal representations of the objects match the ones in the compiled code.

Dynamic encapsulation occurs at run-time, and allows the compiled code to exist without the knowledge of any particular object representation. Abstract base classes with C++ virtual functions support a limited form of dynamic encapsulation, but only for objects originally designated to inherit from those classes. Some languages, such as Smalltalk, support dynamic encapsulation, but with significantly less performance than statically encapsulated languages.

An object model using dynamic type-binding is presented which allows the flexibility of dynamic encapsulation with much of the efficiency of static encapsulation. With this model, objects can potentially communicate and migrate across address space and network boundaries without specific prior knowledge of representations, and can invoke functions on local objects with no more run-time overhead than standard C++ virtual function calls.

This dynamic encapsulation model is incorporated into DC++, a C++-based language with extensions that allow for the dynamic encapsulation of existing C++ objects, and DECO (the Dynamic Encapsulator of C++ Objects), a utility for converting DC++ source code into C++.

Key Words and Phrases

Dynamic Encapsulation, Objects, Object-Oriented, C++, DC++, DECO, Distributed, Programming Language, Operating System, Compiler, Type Binding, Delegation, Forwarder

Computing Review Subject Codes

D.3.0 Programming Languages - General

D.3.3 Programming Languages - Language Constructs and Features

D.1.5 Object-oriented Programming

Permission to Copy

Title of thesis ______Dynamic Encapsulation of C++ Objects______________________________
_______________for Distributed Object-Oriented Systems_____________________________
____
____

I, Frank E. Barrus, hereby grant permission to the Wallace Library of the Rochester Institute of Technology to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: ____________ Signature of Author: _____________________________

Table of Contents

Figures

Tables

Examples

0

Introduction and Background

Encapsulation allows collections of data and functionality to be viewed computationally and conceptually as a single entity, called an *object*, by sealing off the internal composition and providing an interface for accessing the object via *methods*. Static encapsulation provides this view at compile-time, but allows users of the object to directly access the internals at run-time, thus breaking the encapsulation and creating dependencies on the implementation. Dynamic encapsulation eliminates these dependencies and maintains the encapsulated view by providing a run-time interface that does not require any implementation knowledge in order to use an object. This has the distinct advantage that the implementation of the object may be changed at run-time without requiring recompilation of the code for users of the object.

Dynamic encapsulation can be used to simplify the process of developing a distributed system. With existing RPC methods, communication between different address spaces is possible, but the programs on each side are required to have a full understanding of the objects being passed across the network. This is accomplished by sharing header files that describe the structural layout of the objects. [*Bloomer92*] With dynamic encapsulation, any object can be replaced with a proxy object at run-time, allowing transparent networking of objects even for systems designed without networking in mind. This technique can be used to turn a locally-developed set of interactive objects into a distributed set of objects without requiring any re-implementation, or even recompilation, of the objects or the code that invokes them. Also, for objects that are migrated, all programs that access the object can still have a consistent view, even though only one program is accessing the actual object and the rest are calling through proxies. Dynamic encapsulation will be discussed in more detail in *Section 1 (Object Theory)*.

C++ provides some of the features of object-oriented programming with most of the efficiency of C, but some important aspects of a fully dynamic object model are missing, and some aspects of the model are implemented in ways that can make them potentially invalid. Virtual member functions provide some of the missing support, but their usefulness is often limited by their means of specification. [*Joyner96*] C++ also suffers from assuming that the implementation and interface hierarchies are the same. [*Martin93*] *Section 2 (The Existing C++ Model)* will look into this further.

Truly dynamic objects, whose methods and attributes can be looked up at run-time and whose interface types can be dynamically checked, would allow different machines to communicate without having any previous knowledge of implementation details. As long as the expected functionality is available (as specified by interface types), various implementations of objects would remain compatible, regardless of changes in the location or representation of the data. Such a model is proposed in *Section 4 (Dynamic Encapsulation Model)*.

0.1 Motivation

The majority of the original research for this thesis came from the development of the second major version of the ShagOS [*] operating system, [*Barrus96*] and eventually led to the development of DECO, the *Dynamic Encapsulator of C++ Objects*, and a set of dynamic enhancements to the C++ language, known collectively herein as DC++.

0.1.1 *Shag/OS*

The Shag/OS operating system, (now written as ShagOS for newer versions), was one of the original reasons for adding dynamic extensions to C++. This was an object-based operating system written in C, [*Barrus92*] and was the core of many of the ideas developed in this thesis. It used many complicated macros and pre-processing to accomplish what C++ can now do much more cleanly.

In the original Shag/OS, objects were created from hand-written tables that associated attribute identifiers with their values, and method calls required hand-written code that matched the method identifier with the appropriate action. This resulted in a fairly flexible object system, but it was not straightforward to program in; too much specific knowledge related to the underlying object system had to be explicitly written into the code. This meant that changes in the implementation of the object system required not only recompilation, but also rewriting all of the code that declared objects.

0.1.2 *ShagOS, Take Two*

When work first began on a new version of ShagOS written in C++, [*Barrus95*] it was mistakenly assumed that the object-oriented nature of C++ would map cleanly to the object-oriented operating system model, thus hiding many messy details and eliminating the need for any macros or pre-processing. However, this turned out not to be the case, since C++ provided a mostly static object-oriented model, and ShagOS required a fully dynamic one.

The addition of dynamic support to the C++ language resulted in the creation of a new language, tentatively named DC++ for the purposes of this paper, that consists of all of the standard C++ features, plus extensions to automatically create and invoke dynamic classes and types. Initially, it appeared as if compiling the entire DC++ language would be needed to add the new syntactic constructs to the expressions used for calling methods and accessing attributes. Eventually ways were found to work this into the existing C++ syntax, so that extensions were only needed for the declarations of dynamic classes and types. DC++ header files can now be compiled into C++ header files, which can then be included into standard C++ code. However, compiling an entire DC++ program, rather than just the header files, allows for a simpler syntax for dynamic calls, especially those used for dynamic type binding. This will be discussed in more detail in *Section 5 (The DC++ Programming Language)*.

Since dynamic object-oriented communication, including calls between objects in different address spaces, or on different machines, is the glue that holds the entire ShagOS operating system together, the development of the operating system has served as a test-bed for the DC++

[*] This operating system was named ShagOS because my college nickname at the time was "Shaggy" and I had previously written a terminal emulator that people had started calling "ShagTerm." The next logical step seemed to be to call my experimental operating system "Shag O.S."

object framework and the DECO compiler. Thus, many of the program examples contained herein are from the operating system.

0.2 DECO

DECO extends the C++ syntax for class declarations to provide dynamic classes with type information known at run-time. In its current implementation, it translates DC++ (Dynamic C++) † code into conventional C++ code that, together with the DECO libraries and standard include files, provides an object framework with dynamic inheritance, method selection, and dynamic construction of classes, as well as run-time information about classes and types.

Since DECO converts DC++ to C++, it should be possible to manually implement all of the dynamic extensions directly in C++ without using DECO. However, doing so would generally be exceedingly tedious and error prone and would require manually changing all of the class and type descriptors ‡ if the implementation of dynamic objects changed. DECO shields the programmer from such implementation-specific details, and instead allows program development to be in the realm of more abstract object concepts.

0.3 Goals

The main goals of this thesis are to:

- Define an abstract theoretical model of computing with object technology.
  [*Section 1 (Object Theory)*]

  — Show how existing C++ features map onto this model.
  [*Section 2 (The Existing C++ Model)*]

  — Demonstrate the shortcomings of C++ with respect to this model.
  [*Section 2 (The Existing C++ Model)*]

  — Discuss ways of extending C++ to support the complete abstract model.
  [*Section 3 (Extending C++)*]

- Develop a language-independent computational object model, the Dynamic Encapsulation Model (DEM) that encompasses much of the abstract model in an efficient manner.
  [*Section 4 (Dynamic Encapsulation Model)*]

  — Present the syntax of the DC++ language, which contains extensions to the C++ language to support the DEM
  [*Section 5 (The DC++ Programming Language)*].

† Originally, this was going to be called "Dynamically Encapsulated C++", but it seems that a corporation and a government agency have both laid claim to the associated acronym already.

‡ The DC++ compiler (DECO) automatically generates descriptors with information about the layout of instance data for a class and the methods for a type. Although these could be created by hand, it would be a significant amount of work for any sizeable system.

- Describe the use of DECO, which provides the support for DC++, and discuss the implementation of DEM used by DECO, along with potential alternatives.
  [*Section 6 (DECO)* and *Section 7 (Implementation)*].

  — Discuss some applications for DECO and the DEM model.
  [*Section 8 (Applications)*]

0.4 Conventions

In an attempt to aid readability, the following font-style conventions will be used:

- Bibliographic references, as well as references to terms, figures, and examples are printed in *italics*.
- Where terms are defined, they are printed in **bold**.
- Section titles, and the names of programs or products are printed in Helvetica.
- Source code, keyword references, and identifiers used in code or models are printed in `Courier`.
- Filenames are printed in *`Courier Italic`*.
- Examples of user input are printed in **`Courier Bold`**; this is also used to emphasize changes made to a file.

0.5 Intended Audience

This thesis assumes that the reader has an understanding of the concepts used in imperative programming languages, especially C, as well as some basic familiarity with C++. Prior knowledge of object-oriented programming concepts and languages should not be absolutely necessary, although they should certainly prove beneficial as long as the reader is willing to keep an open mind to variations in techniques and terminology.

As an aid to understanding this document, a glossary is provided starting on page 199. For additional background, the bibliography, which starts on page 209, is followed by a list of references organized by topic.

0.6 Organization

This thesis takes a somewhat iterative approach to presenting its material. Some of the same topics are revisited in each chapter, with successive chapters providing more details about each topic. One of the reasons for this is because many of the concepts are very closely intertwined, and it would be difficult, if not impossible, to present the reader with each concept in its entirety without first building an understanding of the relationships between the concepts. An attempt was made to organize the material in as much of a linear fashion as possible, but there are many areas where several topics depend on an understanding of each other, so there was no obviously correct order in which to present the material. Therefore, in order to reach a complete understanding of the material, it may be necessary to re-read some of the earlier sections after reading later sections. For this purpose, many cross references are given throughout this thesis, and there is also an index starting on page 217.

1

Object Theory

This section will describe the necessary object theory for defining the Dynamic Encapsulation Model, which will be presented in *Section 4 (Dynamic Encapsulation Model)*.

1.1 Terminology

Some of the terms used in this document may differ from standard terminology, but part of this is due to the fact that "standard" terminology itself seems to change depending upon whose research is being examined. The following information is being presented now to help clarify some of the terms, but each will be explained more thoroughly later in the document.

- The basic unit of organization, which everything is considered to be, is an **object**.
- A **token** is used within the computational system to uniquely [¤] identify each object.
- The **features** of an object are all of its externally visible characteristics. Features are divided into *attributes* and *operations*.

  — Attributes are passively observable characteristics of the object.

  — Operations are actions that can be performed on the object, which may in turn cause the object to invoke actions on other objects. Operations cause the state of some part of the whole system to change (this is not restricted to just the *fields* of the object).
- **Methods** are routines used to access an object. They include both the functions used to implement *attributes* and the routines (functions or procedures) that implement *operations*. [◊]
- The **signature** of a function, method, type, or class describes its syntactical compatibility.

¤ Tokens only need to be unique to the extent required to get the correct results when using that object in computations. Hence, a token need not convey all information about an object, and may be allowed to to be ambiguous if all the objects it could refer to are equivalent with respect to the operations that will be performed on the object using that token.

◊ Some researchers use this just for the operations, so there is some confusion surrounding this word.

- The **behavior** of an object consists of a description of how the *attributes* of the object change with respect to *operations* performed on it. If other objects are also affected by the operations, then these changes, known as *side effects*, are also considered part of the object's *behavior*.
- **Fields** are variables stored for each instance of an object as part of its implementation. There may or may not be a simple mapping between *attributes* and *fields*. Fields are also sometimes referred to as **slots**.
- Objects are organized according to *classes* and *types*, which are not identical concepts, although some object-oriented languages, such as C++, treat them as such.

  — A **class** describes the precise behavior of an object, defines the fields needed to model this behavior, and generally utilizes direct manipulation of the representation of *tokens*. A class may statically or dynamically implement one or more *types*.

  — A **type** describes the generalized *behavior* and *features* of an object. No actual implementation is directly associated with a type. A *type* can be used to classify objects, and in this sense it can be seen abstractly as a set containing all objects that are of that *type*.
- **Inheritance** is any means of reusing features of a parent object in a new object without having to explicitly copy them. Sharing and implicit copy semantics are both considered forms of inheritance.
- **Genericity** is a means of taking a generic object and reusing it as a more specific object by further constraining its *types*
- **Static** generally means fixed at compile-time. Once a program has been converted to machine code, all aspects of it denoted as *static* cannot change.
- **Dynamic** generally means that a specified aspect of the system is flexible at run-time.

There is also a glossary, which starts on page 199, with a much more extensive list of terms.

1.2 Objects

Every conceivable entity or concept can be considered an **object**, including the classifications used to organize sets of similar or related objects. Values, such as simple numbers, are also objects; they represent the concepts of entities that behave in a particular manner with respect to mathematical operations. The concept of *all integers* is also an object, although it acts as a **meta-object**, which can be used for organizational purposes and for generalizing the workings of a computational system. **Aggregates**, [¥] created by joining multiple objects together, can be seen either as a single object, or as a grouping of all the individual objects. Likewise, sets are objects, and the individual elements of the set may be seen separately as well.

An object can be thought of conceptually as having *attributes*, which describe its properties including its current state, as well as *operations*, which can be performed on it and cause state

¥ These are sometimes also known as **structures** and **records**. However, these terms often carry other implicit meanings with them. For instance, in CLU, *structures* are immutable, but *records* are mutable. [*Liskov81*] changes. In a more concrete sense, an object can have *fields*, which store the state information required for the implementation.

1.3 Classification

Classification is a process of dividing objects into sets and subsets based on similar properties. Every set has some defining characteristics that hold true for all of its members. Every object can belong to zero or more sets, and may even dynamically change its membership if some of its properties change. [@] Sets themselves can also be classified, forming a meta hierarchy of classification. This process can be carried out infinitely to whatever degree it is considered useful to do so.

The human mind is constantly classifying objects it encounters, based on their attributes and behavior. It abstracts specific knowledge into general concepts, and even classifies the concepts, thus dealing with meta-knowledge. Pattern recognition and the ability to classify objects and concepts based on similar patterns seem to be essential to understanding and dealing with new situations.

Every step of learning or self improvement is a changing of previous conceptions, by replacing them with new ones, or dropping them altogether. Some ideas that once seemed different are joined, while others that seemed the same are now seen to be different, thus building a mental model that more accurately represents the complexities of the ‘‘real’’ world in a more simplified way, or perhaps a way that is just more appropriate due to changes in the surrounding world.

Likewise, a computer program, in attempting to solve the problems put to it, is working to redefine the abstract models it has that represent the world of the problem, rules, and the solution. Bit by bit, complex abstract object representations are replaced with simpler, more concrete ones until a solution is reached. In most cases, the various levels of abstraction and refinement necessary to solve a problem are all handled in advance by the programmer, so that the final program only handles a very specific set of problems, without requiring much, if any, abstraction or classification to be done at run-time.

Technically, the entire outcome from any system is predetermined if no external inputs are used. From any given start state of the computer, only one final state can possibly exist. Even randomness must be mathematically predetermined if it cannot come from an outside source. Thus, for a given computer system and any fixed pattern of external input states, there is a direct mapping of input states to output states, and a finite state machine can be developed that clearly shows what state the computer will ultimately end up in, and what the outputs will be. If timing is significant to the computation, then some form of clocking information will also need to be added to the inputs.

Because of limited resources, many computations may have to be performed to arrive at that final state, but with enough memory a direct table could theoretically be created that brings the system, or an individual object in the system, directly to its final state, given the full set of

[@] Failing to change the classification of an object when its properties change can be a significant source of bugs in dynamic systems and can cause a lack of proper adaptation in human intelligence as well.

inputs at once. Of course, to any outside observer, the final state is irrelevant; all that matters is that the system also responds with the correct outputs, but this too could all be directly mapped.

At first, then, it would seem that the classifying of objects into types is solely for the purpose of simplifying all the possible mappings, so that generalized rules can be used to compute the results, rather than having each mapping explicitly written. However, returning to the analogy of the human mind, it should be noticed that classifying objects based on abstract concepts is what allows the mind to also deal with objects it has never encountered before, and to make intelligent guesses as to how these new objects can be used, based on observable properties. Of course, the mind also makes mistakes, because the classification system is based on observation, and always has incomplete information.

Computers, on the other hand, are generally only capable of solving the specific types of problems they were programmed for, and must be reprogrammed for new abstractions. However, by including some meta-knowledge in the programs themselves, it should be possible for programs to adapt to new objects they were not originally designed for, as long as these new objects have the expected behavior or can be queried about their behavior in ways that the programs can adapt to. It is important that the expected behavior is guaranteed under these conditions, because it is considered unacceptable for a computer to make mistakes the way the human mind does.

Forming *objects* from both physical entities and abstract concepts, and classifying these objects based on their attributes and behaviors, forms the foundation of object-based computing models. Adding hierarchies, from which knowledge of behavior can be derived, extends object-based computing into object-oriented computing.

1.3.1 *Classes*

Classes provide a way of creating a blueprint for the generation of objects that share the same general structure and behavior. A **class** specifies the implementation for the objects that are known as *instances* of that class. Every class has its own *signature*, which represents the interface to the implementation. The class may also choose one or more *types* to statically bind to the implementation, providing a direct implementation of those types. However, additional types may be indirectly bound to the class at run-time, or even to individual instance objects, by using the available types to provide the implementation for new types. This *indirect binding* allows a system much more flexibility at run-time than was originally designed into it.

A static class has a fixed representation and obtains a fixed region of memory upon creation. It supports a known set of types, and has implementation functions for each. A dynamic class, on the other hand, is much more flexible, and although it may have a standard representation, the location of its instance data may be moved at run-time, possibly even to other machines, and can also be dynamically restructured. The types it supports can be dynamically changed. This is accomplished through the use of an *object descriptor* that contains information about the class of the object, the location of the data, and references to objects responsible for accessing and managing that data.

1.3.2 *Types*

Types are a means of organizing objects based on their general *behavior*, regardless of their implementation, or precise specifications. Types describe the *features* of an object, but are not concerned with the *fields* stored in the object, or the means of implementing the *operations* or examining the *attributes*.

If two objects have the same *type*, or if the type of one object is a *subtype* of the other object's type, then one object can be substituted for the other. Using types to determine compatibility, rather than implementation classes, allows much greater code reuse and flexibility in a system.

Figure 1-1 shows the classification of some simple objects into different types using dashed boxes. Note that an object of an inner type is also a member of its outer enclosing types. For instance, all circles shown in *type_circle* are also of *type_ellipse* and *type_shape*, and all ellipses in *type_ellipse* are also of *type_shape*. That means that all features of *type_shape* are also features of *type_ellipse*, and all features of *type_ellipse* are also features of *type_circle*. In this sense, types act like sets that define features that all their members must have.

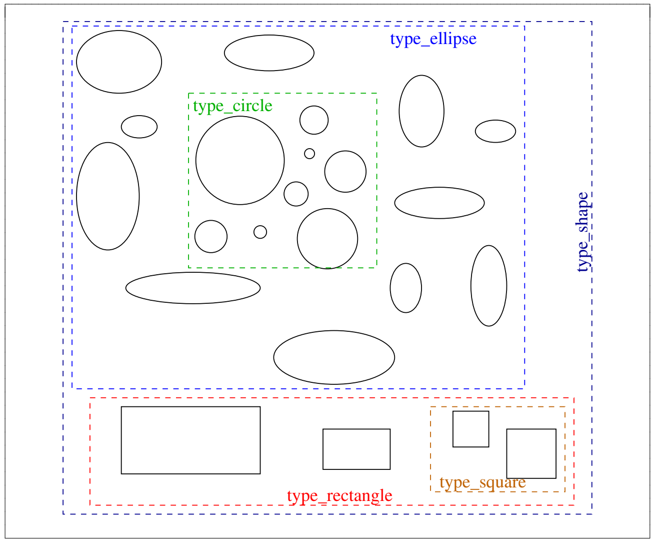

Figure 1-1: Shapes and their Types

Types can also be used to simplify the naming of *features*. To avoid conflicts every feature should have a unique identifier within a given scope. Some languages, such as CLU [*Liskov81*] and BOPL [*Palsberg94*] , consider the feature name sufficient to uniquely identify it for the purposes of type checking; however, in practice it could be exceedingly difficult to ensure that every distinct feature gets a unique identifier when the objects are created by different programmers, and especially when they come from different organizations. Even with objects created by a single programmer, a method named '*add*' or '+' might be used for both strings and numbers with

entirely different meanings. The method or operator name is *overloaded* to have different meanings in different contexts, and that implied context needs to be clearly stated in the program. By associating the features with particular types, the required context can be provided, and the unique feature identifier can be considered as the cross product of the simple feature identifier (its name) and the most general type that supports the feature. That is, $Identifier \times Type \rightarrow UniqueFeatureID$.

1.3.3 *Distinction of Types and Classes*

Much of the literature about object-oriented programming assumes that *types* and *classes* are identical concepts. Programming languages like C++ and Eiffel also join the two concepts and use the *class* as the basic unit of typing. There are, however, researchers who insist that types and classes need to be kept separate. William Kent says, "**Type** is associated with the external semantics of objects. Instances of the same type share a common operational interface. **Class** is associated with common implementation. Instances of the same class share a common implementation. Several classes may implement the same type." [*Kent89*] Peter Wegner follows similar reasoning by defining **types** as "implementation independent behavior specifications", and **classes** as "implementation-dependent specifications by code." [*Wegner87*] These are the definitions that will be used in this thesis.

Types and classes are not the same thing. Two objects could both be of type `t_string` and thus act like a string, and support string operations, but yet not be instances of the same class, since they might have different forms of storage and implementation. Classes, however, always implement a specific set of types, although not all types are applicable to all instances of the class. For instance, although all instances of the class `Ellipse` are of the type `t_ellipse`, only some may be of the type `t_circle`.

With the multiple meanings of the words *type* and *class*, there is bound to be some confusion as to the interpretation that is intended. To try to clarify the situation, it might be useful to think of the term *type* in this thesis as meaning *abstract type* since it just represents the *signature* (interface) and intended behavior of the object, without any care for the implementation. *Class* refers to the classification of the implementation. When discussing C++, the term *class* may include the interface as well, since C++ combines the concepts.

1.4 Object Existence and Identity

In order to construct a computational model of objects, and the relationships between objects, it is first necessary to define what objects are, how they exist, and how they can be identified.

1.4.1 *Objects*

Every concept that exists or could possibly exist is an object. Objects are eternal, in a conceptual sense. In other words, every abstract object can be seen as having always existed, and as continuing to exist through all of eternity. Concrete representations of such objects, however, appear to have much more limited lifespans. Yet, theoretically, it is possible to imagine that every object in the known universe continues to exist through eternity as well, but perhaps is just not currently accessible due to the passing of time, or changes in the state of the universe. Obviously, objects that are not accessible in the current *scope* are not important from a computational point of view.

What *is* important is that such objects are consistent if they are referenced by the current scope in the future, or if they are referenced from other scopes.

All objects, whether they be ‘‘constants’’, ‘‘functions’’, ‘‘variables’’, etc., are eternal, in the sense that they never really change. They are simply an abstract notion of an entity that can be uniquely identified in relationships. It is the relationships that define the changes. Thus, although objects can be said to have a ‘‘state’’, that state is really just a subset of all the relationships in the universe of discourse in which that object lives. In this sense, objects are devoid of substance; they are simply sets of functions that have dependencies on not only their explicit arguments, but also implicitly on the functions of other objects.

Unfortunately, this very abstract view of objects does not map well to the computational devices we currently use, which maintain a notion of state with the values of their memory bits. So, for the purpose of simplifying the descriptions and implementations of objects, all mutable objects referenced in this document shall be considered to have an encapsulated state that can be examined and modified and which affects the behavior of the object. However, it should not be assumed that there is a unique encapsulated object state for every object.

It is possible for several objects to share some or all of the same memory for maintaining state information, and it is also possible for the state of an object to require varying amounts of memory scattered all over the address space. In fact, the current state of an object might depend upon external inputs, such that the state of all of physical memory may not be enough to define the state of the object. Thus, all that can be said about the state of an encapsulated object is that it can be derived from the current state of the universe in which it exists. The scope of what is considered the universe will depend upon the hardware and software being used. Encapsulation hides the need to worry about how this state is maintained, and allows the conceptual entity to be simply treated as a single object.

1.4.2 *Tokens*

Tokens are objects that are used to symbolically represent other objects, such that referring to the token is equivalent to referring to the object it represents. [*Kent91*] Tokens provide the only way to deal with abstract concepts in a computational system, or, for that matter, to deal with computations involving real physical objects that exist outside the computational system. On paper, written numbers, such as ‘1’, ‘23’, ‘47’, etc., are tokens for the abstract numerical concepts they represent, and proper names, such as ‘Tad’, are known to represent particular people, such that they can be referred to. Notice that although numbers may be a fairly universal concept, different representations of numbers can exist, such as binary, ternary, octal, decimal, hexadecimal, etc., and so the number ‘121’ as a token could really represent many different abstract numbers depending on the interpretation. Likewise, ‘Tad’ could refer to many different people. # Without any additional qualifiers, there may be an assumption as to which person this refers to, but in a computational system, where precision is required, such ambiguities are not permissible. Therefore, some means is necessary to ascertain the precise object to which a token refers, or at least one that will yield sufficiently accurate results for the intended purpose of the computation.

Perhaps there may not be many people named Tad in a local scope, since it is not an overly common name, but certainly there are many in the global scope.

1.4.2.1 *Contexts*

A **context** is used together with a *token* to define the precise object to which the token refers. [*Kent91*] This can be written as:

$$Token \times Context \rightarrow Object$$

Note that this function is abstract, meaning that it exists outside the computational system, since it is not possible for any concrete operation in the system to directly yield a real object as a result. This is a simplified form of the work by Kent [*Kent91*] that maps *tokens* (in the computational system) to *symbols* (outside of the computational system) which are then combined with a *scope* (or *context*) to yield a *handle* to an abstract or concrete object, which is then called a *thing*. Since all of these mappings occur outside the computational system and since all functions really yield references to objects, not the objects themselves, it seems only logical to dispense with the extra levels of abstraction, and instead have the *token* in a given *context* refer directly to the *object*.

At any given time or in any given portion of a computational system, only one context is active, although that context may contain other subcontexts. For a token to be *valid* in a given context, there must exist a singular functional mapping from the token and context to a unique object. It is possible for that object to be a set of other objects, but it is important that the mapping is a functional relation and not ambiguous. The set of all contexts in which a token is *valid* and refers to the intended object is the **scope** of that token, with respect to the intended object.

Every cell of memory provides a context for the data contained therein, with this context being assumed to be equivalent by all code accessing that location. If there is a lack of consistency, then errors will result, because the data will be referring to different objects due to a change in context. It is possible for the context of that cell to vary through the lifetime of a program (due to the use of a variant record □ or dynamic memory allocation and reclamation), but at any given time, there must be consistency.

1.4.2.2 *Tokens as Objects*

Tokens exist entirely within the computational system. As such, the tokens themselves can be seen as objects, in whatever fashion they are represented. Hence, the numbers from 0 through 9 could be grouped by a computer into two sets as follows:

$$\{0, 3, 5, 6, 9\}, \{1, 2, 4, 7, 8\}$$

Likewise, the same numbers might be grouped by a human being as follows:

$$\{0, 2, 3, 5, 6, 8, 9\}, \{1, 4, 7\}$$

Both of these ways of grouping are based on the token itself, rather than what it represents. In the first example, the computer used the binary representation of the number, and classified the numbers based on whether they had an even or odd number of ‘1’ bits. In the second example, a standard visual representation of the number was used, and the numbers were classified according to whether or not they had any rounded parts.

Such examples may seem ridiculous at first, but in fact most computations really are performed by using the representations of the tokens. Consider an operation such as 475×3306, with the standard base 10 numerical representation being used in this case. Although both of these tokens represent distinct numbers, few people would have the result to such an operation memorized. If forced to compute the answer by hand in today’s age of computers, most people

□ This is called a `union` in C/C++.

would use a multiplication algorithm they were taught long ago, which uses the individual parts of the token by themselves (the digits) to compute the answer. Seeing the token as simply a representation of an abstract object is fine when tokens are just being passed around between functions, but when an actual computation needs to be performed using one, often the token itself must be seen as an object, such that its traits might be used to simplify the calculation of the result. If this were not the case-- for instance, if 3306_{10} was instead represented by the token $\clubsuit$, and 475_{10} was represented by the token $\spadesuit$, and there were single, atomic, tokens such as these for all possible integers, would it be so obvious that the result of $\clubsuit \times \spadesuit$ was $\heartsuit$? *(which, as it turns out, is a token equivalent to* 1570350_{10}*)* Probably not, since this would require memorizing all the combinations of numbers that could ever be considered for such an operation. It should be noted, therefore, that the base ten numbering system provides unique tokens for every number in such a way that calculations can be done by manipulations on the tokens themselves.

Likewise, on most computer systems, *binary* numbers were chosen to simplify the circuitry used to perform calculations, so that, rather than storing a table of all possible results, calculations could be performed by simply manipulating the *bits* of the binary tokens.

Extending this concept still further, it can be reasoned that every data structure or abstract datatype in a computer program, no matter how large, or how scattered through memory, is a single token, representing some abstract or concrete object. Some mechanism is used to determine the size of the bit pattern, and not all combinations of bit patterns are necessarily *valid*, but those that are can be said to refer to some object, whether or not that object actually exists in our universe. Many different combinations, in the same or in different contexts, could refer to the same theoretical object, but the token representations that are chosen should be the ones that simplify the computations. For instance, to store a token that refers to a conceptual circle of radius 5, with a center at (1,3), the following could be used:

$$< 1, 3, 20 >$$

From such a token it is possible to extract all the information needed to describe the circle, (by dividing the third number by the sum of the coordinates to obtain the radius) but not without a higher computational cost than is necessary. Certainly,

$$< 1, 3, 5 >$$

would be a token that is much more easily usable. Note that for some computational systems, 5 by itself might be an adequate token for the circle, if the coordinates are not needed in any calculations. In fact, *circle* or even just *C* might be sufficient for a token, if the radius will not be needed, or can be derived in some other fashion.

A **token**, therefore, is any object inside the computational system which is used as a reference to another object, which may be inside or outside of the system. Since tokens are also objects, they may be referred to be other tokens. A token may contain any amount of information considered necessary or useful to the computational system, or it may contain no information. Sometimes tokens may contain redundant information that could be otherwise derived, simply to reduce the work required to obtain the redundant value. Tradeoffs of size versus speed have to be taken into consideration to decide on the most appropriate representation for a token. For instance, a token that represents a circle could be specified with just the coordinates and radius as follows:

$$< 1, 3, 5 >$$

or instead, an approximation of the area could be included as a computational convenience, as in the following example:

$$< 1, 3, 5, 78.54 >$$

In terms of memory, this is wasteful, since the area can be derived from the radius. However, this

makes sense if *area* is an attribute that will be frequently accessed and speed is important. By storing it directly, there is no need to compute it. In this case, *area* is an attribute that maps directly to a field. In the previous example, *area* was implemented as a *synthesized attribute*.

1.4.3 *Containers*

Some objects may be seen as **containers**, which have other objects as their **contents**. ⊕ That is, rather than merely existing eternally, such as the number 3, a container also has a notion of its current *state*, or what object it currently contains. Depending upon the nature of the computational system, the contents of the container could be described as a function of time, or as a function of the successive external inputs to the system.

Containers can be considered **intensional** objects, whose contents may change, and yet the container is still the same. **Extensional** objects, in contrast, simply exist, and can never be modified. The number 7, or the set $\{4, 8, 9\}$ are both *extensional*, although they may be contained in an *intensional* container object, and may later be replaced with other extensional objects. *Figure 1-2* shows some examples of intensional and extensional objects. The objects x, y, and z are all intensional objects, or containers, for extensional ellipses. The arrows from each object point to the extensional object that is represented. Notice that each intensional object has a *token* composed of numbers representing the major and minor axes of each ellipse. The storage of each number is also an intensional representation of an extensional object; in this case a number. The arrows looping back on each extensional object show that each such object only represents itself, although they may have features that reference other extensional objects, such as the axes of the ellipse which reference the corresponding extensional numbers.

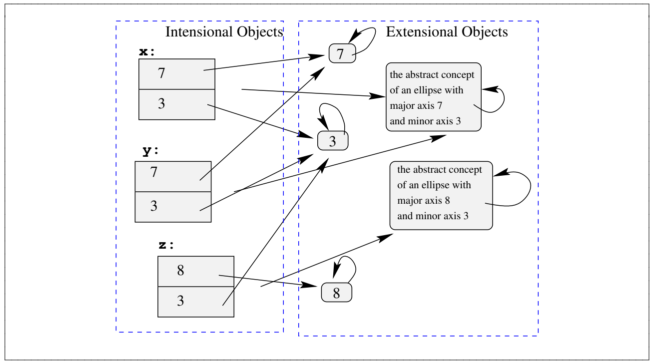

Figure 1-2: Intensional and Extensional Objects

⊕ The basic *container/contents* concept used here is essentially the same as Kent's description of a *carrier* with a *cargo*. [*Kent91*]

The memory storage of a computer can be thought of as divided into **cells**, which vary in size, and which are each used to store one *token*. Cells do not necessarily correspond to traditional memory boundaries such as bytes and words. Every *cell* can be thought of as acting like a *container*, although many of these containers never change their *contents* during the normal operation of the system. In such constant cases, a reference to the container, or *cell*, can be considered equivalent to a reference to the object referred to by the contents of the container. For example, a reference to cell #123, which is storing the token 3 in binary form, can be considered equivalent to a reference to the abstract concept of the number 3, as long as it is guaranteed that the token 3 will not be replaced by another token in cell #123 during the operation of the program.

Generally, tests for equality that involve containers are really tests for the equality of the contents. For instance, given two strings, stored in different *cells*, most equality tests are concerned with whether they both refer to the same abstract string concept, which is represented by a *token* in the container, not by the container itself. Likewise, file comparisons are generally concerned with the complete ordered set of all bytes in the file, not with whether the file occupies the same storage space on the disk. It is important to note this difference, because although equality of containers implies equality of contents, the converse is not true. In fact, the physical representation, or values stored in the cells for a given container, do not need to be identical to that of another container for the stored objects to be considered identical. As an example, the same ellipse could be stored as a major and minor axis, or as a major axis and an eccentricity. *Figure 1-3* shows some containers and the types of objects that can be assigned to them.

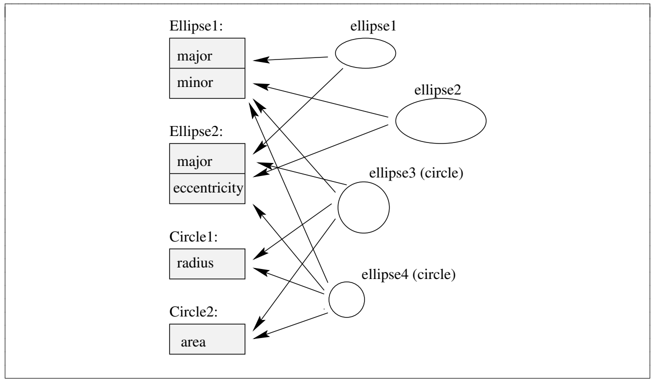

Figure 1-3: Assignment to Containers

1.4.3.1 *Pointers*

A reference to a container is often stored as a **pointer**, which is a token that is constructed of the memory address of the container. Just as with all other kinds of tokens, a *pointer* is not guaranteed to be valid in all contexts. In particular, if that token is passed to another computer, it will have no meaning. Instead, it is necessary to map the pointer to another token that will be a valid reference to the container on the new machine.

It is important to recognize that pointers are not necessarily always used as references to containers. Pointers can also be used to extend tokens, by not requiring them to be in one contiguous piece of memory. Confusion between pointers as being references to new tokens or extensions of the same token often leads to mistakenly interchanging sharing and copying semantics. To avoid this confusion, it must clear whether a pointer is being used to extend a token (in which case the token can be called an *indirect token*), or is being used as a reference to a new token. [*Kent91*]

Tokens may be compound entities, consisting of many smaller tokens that refer to either *intensional* or *extensional* objects. The references to extensional objects can be simple embedded tokens, but the references to intensional objects are references to other containers, and thus are usually pointers, although other implementation-specific encodings are also possible, such as indices into an array, or handles that can be looked up in a table.

1.5 Signatures

Peter Wegner defines **signatures** very succinctly as ‘‘syntactic interface specifications.’’ [*Wegner87*] Signatures of functions, methods, and types are the set of characteristics that can be used to determine syntactical compatibility. It is important to note, however, that this does not by itself imply anything about the semantics. The signature can be used to determine if two entities are syntactically compatible, but some additional mechanism is necessary to test for semantic compatibility. If syntactical analysis is meant to catch semantic errors, then some aspect of the semantics must be incorporated into the signature.

1.5.1 *Function Signatures*

Function signatures consist of the *arity*, or number of formal arguments, of the function, along with the type of each argument, and the return type. Multiple return types are possible, in which case the number and types of returned values should also be part of the signature. Essentially, the signature of a function is the ordered set of all parameters and results, or the specified domain and range of the function.

In general, function signatures have the form

$$P_1 \times P_2 \times \cdots \times P_n \rightarrow R_1 \times R_2 \times \cdots \times R_m$$

where all P_i, representing the domain, may vary *contravariantly* with respect to the signature, and all R_j, representing the range, may vary *covariantly* with respect to the signature. The importance of these relationships will be discussed further in *Section 1.6.3 (Function Substitutability)*.

1.5.2 *Method Signatures*

Method signatures are very similar to function signatures except that the first formal argument, *self*, is not explicitly stated in the function definition. The implicit *self* argument, also known as `this` in C++, [*Stroustrup93*] and `Current` in Eiffel, [*Rist95*] is bound at run-time to the object the method is invoked upon, and therefore its type varies *covariantly* with the signature. This is an exception to the rule requiring function parameters to be *contravariant*, and thus the *Self* type is unique in this respect. It is possible for *Self* to be specified as the return type of a method as well, although this is not exceptional since covariant return values are already standard for all functions.

The common form for method signatures is

$$T \times P_1 \times P_2 \times \cdots \times P_n \rightarrow R_1 \times R_2 \times \cdots \times R_m$$

where the domain consists of T, which represents the type of the object the method is invoked upon and varies *covariantly*, together with all P_i, which vary *contravariantly*. The range consists of all R_j, which vary *covariantly*.

There is another variation of method signatures, sometimes called multi-methods, which would appear as follows:

$$T_1 \times T_2 \times \cdots \times T_n \times P_1 \times P_2 \times \cdots \times P_n \rightarrow R_1 \times R_2 \times \cdots \times R_m$$

Multi-methods involve methods invoked on two or more objects simultaneously, and generally represent interactions of the objects. Due to the complexity often associated with multi-methods, they are beyond the scope of this thesis, but would provide an interesting area to carry this research further.

1.5.3 *Type Signatures*

In an ideal world, type signatures would be composed of a set of named method signatures, along with the required semantics for each method. They can be represented with a set of tuples such as the following:

$$\left\{ \begin{array}{c} < name_1,\ MethodSig_1,\ Semantics_1 >, \\ < n_2,\ M_2,\ S_2 >, \\ \cdots \\ < n_n,\ M_n,\ S_n > \end{array} \right\}$$

Unfortunately, specifying the complete required semantics for each method in a formal system can be extremely complicated, and certainly no simple form exists to do so in a language such as C++. However, it is possible to easily express a more simplified interface signature, by eliminating the semantics:

$$\left\{ \begin{array}{c} < name_1,\ MethodSig_1 >, \\ < n_2,\ M_2 >, \\ \cdots \\ < n_n,\ M_n > \end{array} \right\}$$

Thus, an example of an interface signature for a type that stores an *Int*, and allows it to be read or written to would be:

$$\left\{ \begin{array}{c} < \text{“}get\text{”}, \rightarrow Int >, \\ < \text{“}set\text{”},\ Int \rightarrow Valid > \end{array} \right\}$$

Note that *Valid* is just a special type that can have two values-- either success or failure. Any function or method that can fail can be seen as possibly returning a failure value, which may or

may not be directly handled by the caller of the function. * Also, all undefined functions can be seen, at least in a theoretical sense, as existing functions that just return failure. If returning a failure value is unnecessary, the ''set'' function above could have been rewritten to just return *self* instead.

Unfortunately, this now leaves no way of checking for semantic compatibility. Formally, the best that can be hoped for is well-chosen method names so that each name uniquely specifies the required semantics. This can lead to excessively verbose method names that actually make programs more confusing to read. It is also very difficult to ensure unique names across multiple organizations that may be developing interoperable objects, or to ensure that methods with the same semantics are always given the same name. Informally, comments can be used to clarify semantics, but they cannot be used for automated type-checking. Therefore, the best option seems to be to use the type name to identify the intended semantics of all of its constituent methods.

In many languages, such as C++, the type † name alone uniquely identifies its signature, ‡ which can lead to many incompatibilities when parts of the signature are changed, or if another type happens to have the same name. Luckily, such problems are generally caught when compiling the program, since syntactic incompatibilities usually result. However, it is possible for subtle errors to surface only at run-time, since a particular usage of a class might not exercise enough of its functionality to cause a syntax error when a similar but distinctly different class is mistakenly substituted.

Conversely, there are languages, such as CLU, which use an interface signature alone as a type signature, [*Liskov81*] which can lead to serious complications when an object with completely different semantics, but the same interface, is substituted. Technically, this is a case of failed type-checking, but from CLU's point of view, the interface was enough to validate type consistency. Unfortunately, this kind of error can only be caught at run-time, and can lead to subtle inconsistencies in a system.

A more type-safe solution is to use a signature that is composed of both the name (from which the semantics are implied) and the interface signature. Within a given compilation unit, it should be sufficient to use the type name as the signature, as is done in C++, as long as a more complete identifier is made available and used for checking compatibility between compilation units and between objects in multiple address spaces or on multiple machines. This more complete identifier must be guaranteed to be unique for each unique type. Any change to either the interface signature or the required semantics of any part of the type creates a new unique type and therefore requires a new unique identifier. If the new type is meant to be used where the old one

* Values that are not returned via the conventional return mechanisms are called exceptions, and are not focused upon in this thesis. Future work should incorporate them, since they are a useful way to avoid directly handling all of the possible return values (especially error states) of a function, and their proper use can lead to a much more robust and flexible system.

† Although this thesis draws a line between *types* and *classes*, every class in traditional class-based languages can, in theory, be seen to have an implicit type signature, which consists of all the public methods of that class. For languages that allow direct access to public *fields*, those fields must also be considered part of that implicit type interface. Therefore, for languages that combine the concepts of type and class, this partial abstraction of the class serves as the type, and the name of the class is also implicitly the name of the corresponding type. When referring to these languages, the conceptual term will be used, and thus classes will be called types when used as such.

‡ Within a compilation unit, types are guaranteed to have identical interface signatures (because the declaration had to be included), but when linking across compilation units, types with the same name are considered compatible, even though they could have had completely different interface signatures.

was, then it will be necessary to perform a global inspection of all areas where the old type was utilized, and ''upgrade'' them to use the new type. For local code that recognizes the type only by its name, this upgrading will always be required. Obviously, any change that makes the type identical to a previously generated type should require the use of the previous identifier to allow compatibility.

1.6 Substitutability

A very important concept to object-oriented programming is that of **substitutability**, or the ability to replace an object of one type with one of a different type. This substitution is sometimes also called **subsumption**, [*Abadi96*] although much research also refers to this as the *conformance* of types. It is this ability, to take an object of one type and use it where an object of a different type was expected, that gives object-oriented programming languages more expressive power than languages that require strict equivalence of types.

Some object-oriented programming languages relax the requirement of conformance to the extreme, allowing any object to be *subsumed* to any type. However, without some form of static type checking at compile-time, every operation on the object could potentially fail at run-time, and this might greatly decrease the efficiency of the system as well as introduce the possibility of errors in the design and implementation that are not caught until much later at run-time.

In order to determine the conformance of two types, some basic rules have to be developed to determine if an object of one type can be subsumed to another type.

The **Liskov Substitutability Principle (LSP)** states the following:

> "If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T." [*Liskov87*]

In other words, if S is a subtype of T, then all objects of type S must be usable wherever an object of type T could be used, without changing the behavior of the system. What defines a change in behavior is left up to the designer of the system because strictly following the *LSP*, and allowing no behavior changes whatsoever, would greatly limit the flexibility and code reuse of a system. Thus, instead of requiring the behavior to be exactly the same, it should be semantically the same in terms of the intended purpose of the type.

1.6.1 *Type Substitutability*

In general, a type A can be said to be a subtype of B if every *feature* of B is also a feature of A. This rule can be relaxed somewhat, however, and it can be said that every feature in B must have a corresponding feature in A whose type is a subtype of the one in B. This relaxation still enforces the *LSP*, while allowing more generalized types to be considered subtypes.

Referring back to *Figure 1-1* on page 11 as an example of subtypes, it should be noted that all features of *type_shape*, such as the *area* function, are also features of *type_ellipse*, and all features of *type_ellipse*, such as *area*, *major*, and *minor*, are all features of *type_circle*.

To ensure correctness, it is also important that the semantics of the types are considered, in addition to their signatures. The substitutability rules for functions will be used to ensure this.

1.6.2 *Class Substitutability*

Since *classes* define the implementations of objects, but *types* should be used to interact with objects, there is technically no need to define a *subclass* relationship. However, in the interests of efficiency, it is sometimes useful to access objects directly by their classes. In such cases, a class A can only be substituted for a class B if the entire structure and layout of B is contained within the structure and layout of A, and all of the features of B are supported by corresponding features of A. Notice that this requirement is far more strict than the requirement for subtypes, due to the fact that assumptions are made about the implementation when dealing with classes directly. Unfortunately, this sort of subclass relationship says nothing about the semantic compatibility. Thus, subclass relationships are generally overly strict for the implementation, and not strict enough for the behavior.

1.6.3 *Function Substitutability*

To substitute one function for another, the *signatures* of the two functions first need to be considered.

Functions vary *contravariantly* with respect to their parameters, and *covariantly* with respect to their return types. [*Abadi96*] This means that a substituted function, in order to remain compatible with the original interface, must have parameters that are equivalent to, or supertypes of, the original parameters, but the return value must be of the same type or a subtype of the original return value. ¤ The C++ Committee must have recognized this fact when they relaxed the requirements for return values of overriding virtual functions, however a corresponding relaxation for parameters was not agreed upon. [*Koenig95c*] This is probably due to the ambiguities that could result when determining whether a function in a derived class is *overloading* or *overriding* a virtual function in a base class.

It is not enough to simply match the signatures of the functions. The semantics of the function must be considered, and for functions that are members of classes, the state of the object, and perhaps the state of other referenced or global objects may also need to be considered to determine the semantic equivalence of two functions.

Most forms of semantic specification are informal, generally using comments rather than being written in a format that can actually be used by the computer to verify semantic correctness. [*Blair91b*] A large number of bugs are introduced into programs due to a lack of semantic consistency, and therefore a means of performing semantic type checking could catch many of these errors at compile-time, rather than through run-time testing. Such a means of semantic specification, however, is beyond the scope of this thesis.

1.6.4 *Method Substitutability*

Methods (also known as member functions in C++) follow most of the substitution rules for functions, since they are essentially functions with an implicit extra argument known as `self` (or `this` in C++). Despite the fact that `self` is an argument to the function, submethods are allowed to consider `self` to be a subtype, which varies covariantly instead of contravariantly.

¤ Technically, this is not entirely true. The return value must be of the same type or subtype of the original value only when the arguments are of a type that is equivalent to, or a subtype of, the original parameters. When the function is called with parameters outside this domain, the range of the function need not adhere to any restrictions.

This is due to the special nature of `self`, which by its very definition will be guaranteed to be bound to the object on which the operation was invoked, an object that is always of the same type or a subtype of the one for which the function was defined.

1.6.5 *Container Substitutability*

Containers can sometimes appear to flip around the rules for parameters and return types, making parameter types seem to vary covariantly, and return types vary contravariantly. If the type variance rules are applied to the container, instead of the contents, it can be seen that the container inverts the substitutability rules of the object it contains.

Abadi and Cardelli [*Abadi96*] observe that, given a pair $A \otimes B$ whose components can be updated, the $\otimes$ operator is *invariant*. That is to say that there is no allowable subtype or supertype relation. However, their example only uses the A component to demonstrate this fact. Therefore, these findings can clearly be carried further to demonstrate that given any type A that can be updated, no variable subtype or supertype can be substituted for it. Since A is an entity that can be updated, it can be seen as a *container* for an object of the specified type, rather than just a simple extensional object of that type. Thus the resulting rule is that containers, when their type allows both the reading and modification of the contents, can only be substituted when the type matches exactly. C++ violates this principle extensively by allowing pointers to writable objects of derived types to be substituted for pointers to objects of base types.

Another way of looking at this is to see the "container" as an object that has two methods, `get` and `set`. Seen in this way, the rules of substitutability can be applied to its functions. For two types, A and B, where A is a subtype of B, and two containers, C_A and C_B, the signatures of the `get` and `set` functions can be defined as follows: [◊]

$$C_{A:}\texttt{get}: \rightarrow A$$
$$C_{A:}\texttt{set}: A \rightarrow Valid$$
$$C_{B:}\texttt{get}: \rightarrow B$$
$$C_{B:}\texttt{set}: B \rightarrow Valid$$

By using the rules of subsumption for functions, it can be seen that the function $C_{A:}\texttt{get}$ is a subtype [¥] of $C_{B:}\texttt{get}$, and yet $C_{A:}\texttt{set}$ is a *supertype* of $C_{B:}\texttt{set}$. Therefore, if only the `get` operation is supported, then C_A is a subtype of C_B. If only the `set` operation is supported, then the situation is reversed, and C_B is a subtype of C_A. If two types are both subtypes of each other, then they must be equivalent types. Therefore, container types must match exactly if they can be both read and written. *Figure 1-4* shows a subtype hierarchy for various shape-holding containers that can be read, written, and both read and written. Similarly, *Figure 1-5* shows hierarchies for readable and writable containers that hold various type of numbers.

This concept of a container applies not only to objects with explicit `get` and `set` operations, but also those with such operations implied. Any partial update of the attributes of an object can be seen as a `get` of the extensional representation of the entire object, followed by a `set` with the new extensional representation of the modified object. These semantics must be considered when checking the conformance of two types.

[◊] In these examples, returning Valid simply signifies that the operation is successful. This distinguishes these operations from ones that are not supported, or return other kinds of values. There is an implied means of returning error codes with most functions.

[¥] Technically, this could be seen as a *subfunction* or *submethod*, but subtyping rules still apply as well.

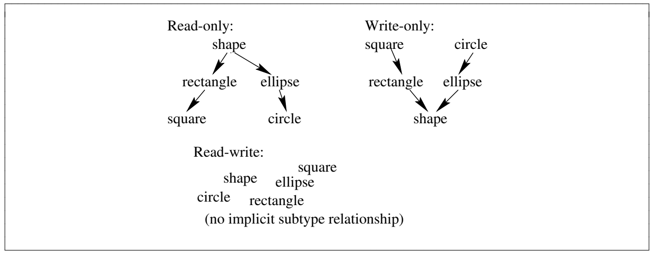

Figure 1-4: Container Hierarchies for Shapes

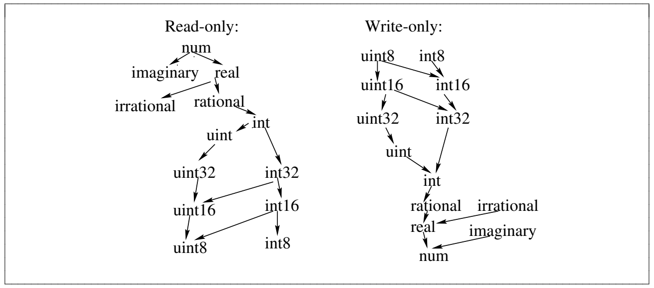

Figure 1-5: Container Hierarchies for Numbers

Requiring container types to match exactly seems rather strict, and does not allow for much flexibility in a system. However, this can be overcome, while ensuring correctness, by separating the `get` and `set` operations for a container into two separate types. By only requiring the necessary operations, functions that only read objects can allow subtypes of those objects, and containers for subtypes, to be passed in as arguments. Functions that only write objects can allow containers for supertypes [@] of those objects to be passed in. When it is necessary to both read and write an object, the only correct option is to make sure the container passed in matches the type exactly. Of course, containers that store supertypes are allowed in such cases as long as the contents are currently guaranteed to be of the required type. This can sometimes be done via dynamic run-time checks.

[@] Note that this refers to supertypes as defined above, and not the C++ notion of base classes. C++ base classes generally cannot be used instead of derived classes for storing objects of derived classes, because the implementation does not support the extra storage needed. However, with a separation of *class* and *type*, a supertype container can be used to store an object of a subtype.

For all subtype relationships, there is a guarantee that an instance of a subtype is also an instance of a supertype, but there is also the possibility that sometimes an instance of a supertype might also be an instance of a particular subtype. Run-time checks can be made on the data within a container to dynamically check if an object is currently of a specific subtype, even though it may not be possible to determine this with static type checking. In such cases, a container for a supertype object may be used even where a readable and writable container for a subtype is needed. It must be ensured that the contents of the container will not be modified by another thread of execution in such a way that the contents are no longer of the required subtype for the duration of its usage.

If the contents of the container are copied to ensure they will not change while being used, it must be realized that a new intensional object, or new container, has been created, which in no way in no way refers to the original container, except for the fact that the current contents are the same as those of the original at the moment the copy occurred. The new object is acting like a snapshot of the original container at a given point in time. If this does not provide the correct semantics, then the container must instead be locked to ensure that no changes are made while this view of the container is being used. During the duration of the lock, copying and sharing the object are equivalent with respect to reading the object, thus the more optimal technique of sharing should generally be used, except in situations where that is not possible, such as when communicating across networks.

1.6.6 *Side Effects*

As a result of executing operations, there may be side effects, or modifications to various parts of the state of the whole system besides just the encapsulated object. These side effects may be the result of operations invoked on other objects that were referenced by the invoked object, or they may be caused by operations that directly modify global state information. Whatever the reason, it is important that side effects are noted in the definitions of operations on both classes and types.

For classes, an understanding of the side effects is necessary to ensure that parts of the system are not getting modified in undesirable ways through the use of a particular class implementation.

For types, it is possible that the side effects are part of the desired behavior of the type, and therefore they should be part of the semantic description of the operations the type supports. If certain side effects are part of the semantics of a type, then all classes that claim to support that type must also support side effects that conform to the side effects of that type.

1.7 Derivation and Reuse

The ability to reuse existing code is one of the major goals of object-oriented programming. This is generally accomplished by taking an existing object and *deriving* a new object from it. Features from the original object may be shared, or removed or replaced, and new features may be added. *Generalization* can be used to factor out common features and create new common base objects from which to derive various objects using *specialization*. *Inheritance* allows features of the parent to be directly shared, while *genericity* and *class substitution* allow customized specializations of the parents' features to be incorporated.

1.7.1 *Generalization*

Generalization is the process of taking the common *features*, *fields* and *behavior* of two or more objects, and creating a new object from the abstraction of these common components. The common features can then be removed from the individual objects, which can now access those features by sharing the new generalized object, using some form of inheritance. For generalizing object instances, this sharing can be be done via *delegation*, and for classes and types, *structural inheritance* can usually be used. Sufficient generalization can be used to completely eliminate fields from classes and thus generate *types* from the remaining attributes and operations. Generalization can also be used to further refine complex types into their basic components, in order to create reusable code.

Generalization is usually more of a design process than a feature of a language. However, it is conceivable that a language could have support for automatically creating generalized classes or types from a given set of classes or types, by picking out the common components, and either eliminating those with non-common dependencies, or including all of the dependencies as well.

1.7.2 *Specialization*

Specialization is a process of refining the behavior of an object by adding, re-defining, or restricting previous behavior. [*Blair91c*] In general, specialized classes are seen as *subclasses*, and specialized types are seen as *subtypes*, however not all forms of specialization are applicable to simple subtyping or subclassing. For instance, a circle is a specialized form of an ellipse, with the restriction that the major and minor axes must now be equivalent, but most subclassing rules will not allow a full definition of a circle to act as a subclass of an ellipse, due to the fact that previous properties are now restricted rather than augmented. Most subclassing systems only handle specialization when it means adding features or possibly re-defining existing behavior in a way that extends the functionality.

1.7.3 *Inheritance*

Inheritance is the ability to make a new object obtain the *features* of a previously existing object without explicitly copying those features. [#] The object that inherits the features is the **subobject**, and the parent object is the **superobject**. These terms carry over to different forms of conceptual inheritance, and thus there are relationships of *subtype/supertype* and *subclass/superclass*. In C++, a subclass is referred to as a **derived class**, while the parent, or superclass, is called a **base class**.

It is quite common to organize the relationships of superobjects and subobjects into some form of **hierarchy** that shows the relationships of the objects with a directed acyclic graph. For instance, *Figure 1-5* shows type hierarchies for types representing readable and writable containers for numbers. These figures follow the standard practice of showing subtypes below their corresponding supertypes.

[#] Usually this is done through sharing, but implicit copying is sometimes used as one way of implementing inheritance.

1.7.3.1 *Structural Inheritance*

Structural inheritance is a form of inheritance whereby all of the structure of a *superobject* is copied into the *subobject*. For classes in C++, this means that the data structure represented by the *derived class* contains all of the members of the *base class*. In a sense, the member functions can also be seen to be structurally inherited, since all of the member functions are carried over into the new derived class. However, the code for them is shared, so in that sense they act more like they were inherited via delegation.

1.7.3.2 *Delegation*

Delegation is a way of sharing that is not limited to extending the structural implementation of instance data. Instead, any calls to methods not present in the current object get passed off to the delegates, which act as if they were *superclass* implementations. [*Gallagher91*] By utilizing delegation, along with a flexible method invocation scheme, it is possible to create objects whose functional and data inheritance can vary dynamically.

1.7.3.3 *Mixins*

Mixins are incomplete implementations that define a set of operations that are usually related to a particular feature. They are designed to only be inherited from and cannot be directly instantiated. [*Snyder87*] The purpose of a mixin is to add functionality to existing classes, by overriding or extending the current functionality. When used at the instance level, by modifying the functionality of an instance with a mixin, this is sometimes called **enhancement**. [*Gallagher91*]

1.7.4 *Genericity*

Genericity is a form of specialization that allows the creation of a new *class* or *type* that is identical to a previous one in function, but with a narrowing of some or all of the types used in the interface and implementation. For example, suppose a class exists that stores lists of arbitrary objects. Genericity can be applied to use the existing list class to create one that stores lists of numbers. In principle, genericity can be applied again to further refine this new class and create one that stores only lists of integers. Unfortunately this second example is not possible with many object-oriented languages, such as C++ and Eiffel, [*Palsberg94*] since their implementations of genericity utilize a notion of a *class template* [*Stroustrup97*] (called a *parameterized class* in Eiffel) [*Rist95*] in which precise types can be substituted to generate a class. Due to this dichotomy of classes and templates, genericity cannot be repeatedly applied in such languages.

In a pure theoretical sense, genericity and inheritance are completely orthogonal ways of reusing existing classes. Palsberg and Schwartzbach contrast them as follows:

- "Genericity: the new class yields the *same behavior* but *new invariants* for that behavior."
- "Inheritance: the new class yields *new behavior* but with the *same invariants*." [*Palsberg94*]

The most useful model of genericity is thus one that considers it on the same level as inheritance. That is, new classes and types can be constructed by taking existing ones and changing their *invariants* (type constraints) without modifying or extending their code, just as inheritance allows new classes and types to take the definitions of existing classes and modify or extend their behavior. Normal subtyping and subclassing rules should also apply to new types and classes created through genericity. In this way, genericity becomes a powerful tool for creating new specialized types and classes with a high degree of code reuse. This technique of genericity is used by the BOPL language and is called **class substitution**. [*Palsberg94*]

1.8 Binding

Abstractions are very useful for expressing an algorithm in a way that does not depend upon a particular implementation, but before such an algorithm can be utilized, various *bindings* must occur, to associate abstract interfaces with concrete implementations. **Method binding** is the process of associating a method name with the appropriate implementation for a particular invocation. [*Blair91a*] Likewise, **attribute binding** associates some attribute with its corresponding storage. This is usually done by using special methods for accessing the attribute, and then using method binding on these *accessors*. **Type binding** carries method and attribute binding a step further by associating all features of a *type* with their implementations, usually by associating the type with a *class* that implements it. In this thesis, most discussions of binding will refer to type binding, although method binding is also implicitly required in order to bind types.

All of these forms of bindings can theoretically be accomplished either statically or dynamically. Sometimes the programmer is given the choice, and sometimes the language will dictate what type of binding will be used.

1.8.1 *Static Binding*

Static binding is performed at compile-time, and thus requires that all information needed to correctly match an interface with an implementation is available at that time, including the types and classes of most objects involved in the binding. □ Static binding generally results in method invocations that are about as efficient as simple function or procedure calls. In addition, since all type information has already been determined, there is no need to handle '`method not found`' errors at run-time. Unfortunately, static binding is also not very flexible, because it requires that all objects are of a fixed type and class that are known in advance. [*Blair91c*]

Depending upon the usage of static binding, it may still provide a high degree of abstraction in the source code, but once the code is compiled, that abstraction is gone, and therefore large amounts of code may have to be recompiled just to make small changes to the implementation, depending upon what is changed and how broadly it is used.

1.8.2 *Dynamic Binding*

Dynamic binding occurs at run-time, and requires that information is available to associate a method selector with the code to execute the method. It is far more flexible than static binding, since no particular knowledge is needed when compiling. However, this also means that error checking is difficult to perform at compile-time, and thus run-time errors such as '`method not`

□ The types of some objects may be ignored if they are not necessary to disambiguate between the available methods. If a type is ignored at compile-time for this reason, then a validity check should be performed at run-time.

`found`' may occur. One possible way to avoid this is to still use *types* and perform *type binding* rather than just using *method binding*. The model in *Section 4 (Dynamic Encapsulation Model)* uses this technique to provide flexibility while maintaining type safety.

1.8.3 *Static Type Checking*

With static type checking, all type information must be available at compile-time. This does not mean that any knowledge of the implementation is required; in fact, static type checking can work quite well with dynamic binding. Since types are checked at compile-time, errors can be caught early, and many bindings based on types can also be performed in advance.

1.8.4 *Dynamic Type Checking*

When dynamic binding is used, but not enough type information is available at compile-time to perform static type checking, dynamic type checks must be performed. Dynamic type checking provides a lot more flexibility to a system, but also makes it more difficult to catch errors. Since types may be found to be incompatible at run-time, unrecoverable errors can occur at that time, but may not always show up under normal circumstances.

Recovery code needs to be provided to handle the case where the type does not match. Sometimes this is expected to occur often and sometimes this is an exceptional case that was not expected, but must be accounted for in case there are hidden bugs in the code. Some languages provides a conditional assignment operator for easily performing run-time dynamic type checks during assignment. This operator is written as '`?=`' in both Eiffel [*Rist95*] and the DC++ language, which will be discussed in *Section 5 (The DC++ Programming Language)*.

1.9 Encapsulation

Encapsulation is a way of providing a simplified interface to an object while sealing away the object's complexities of implementation. To encapsulate is to "join together one or more objects, as if in a capsule, no longer providing direct access to any of the internal objects, but instead providing an external interface so that the entire capsule can be seen as a single object." [*Blair91c*] Encapsulation can reduce bugs and promote reusability [*Flanagan96*] by allowing the code that interacts with an object to stay constant when the internal representation and implementation of the object is changed. It also allows many different objects to share a common interface, and thus be used interchangeably, although their implementations may vary greatly.

Encapsulation is sometimes seen as a way of hiding details from the programmer, so that objects can be used without knowledge of how they are represented in storage. If the hiding of information is not complete, and direct access is allowed to some internal part of the "capsule", then the encapsulation is broken and subtle errors can result when changes are made. Encapsulation can really only be effective when access to hidden information is prohibited by the programming language, thus enforcing the abstraction. [*Pratt84*] Exposing instance variables is one way in which the ability to change an object's representation is reduced. Many implementations of inheritance also compromise strict encapsulation, since they allow descendants to access the internals of a parent object. [*Snyder87*]

Blair [*Blair91a*] provides a much more systematic view of encapsulation, citing it as having the following characteristics:

- representation; the internal data structures used to model and record state
- operations; interface to procedures and functions that manipulate the internal state
- algorithms; implementations of the procedures and functions that are defined
- attributes; name to value pairs that enhance the description of the object
- constraints; limitations imposed on behavior, often in the form of assertions on input, output, and states
- triggers; asynchronous actions that are fired as a result of a change in state

In this thesis, encapsulation will provide a way of hiding the internal implementation of an object and accessing it via one or more *type* interfaces. The type will provide the abstract specifications for operations and attributes that are supported, and the class will specify the representation and algorithms used for a given object. Constraints and triggers will not be directly supported by the work in this thesis, although nothing prevents them from being implemented.

1.9.1 *Static Encapsulation*

Static encapsulation provides an encapsulated view of objects prior to compilation, thus allowing object implementations to be drastically changed without rewriting the source code for the users of the objects. However, once the code is compiled, the particular implementation that was encapsulated is now frozen. It may be possible to change some limited parts of the implementation without recompiling all the code that calls it, but in general the interface and implementation are closely linked at compile-time. The abstraction provided by the interface is only an abstraction for the sake of the programmer, not the final executable. The compiler breaks open the encapsulation, and creates dependencies in the object code that are directly linked to internal aspects of the object's implementation.

For instance, in C++ it is necessary to make the full definition of an object's class available to all code that invokes the object. Even if all accesses to the object are via non-inline functions, as opposed to directly accessing the *fields* of the object, the knowledge of those fields is still used to determine the total size of the object, and knowledge of this size is embedded in the compiled code. Dependence on this knowledge limits the flexibility of code using static encapsulation.

1.9.2 *Dynamic Encapsulation*

Dynamic encapsulation moves the encapsulated view of an object into the compiled code, rather than limiting it to the source code. This means that every aspect of the implementation of an object, including the size of the storage area required, can be changed, and the code that invokes the object will not need to be recompiled, since the encapsulated view is still available at run-time. It also means that entirely new objects can be introduced at run-time, and then encapsulated by providing them with an abstract interface. These objects can then be invoked by any code that was previously compiled to use that interface.

To provide this encapsulated view at run-time, it is necessary that no knowledge of any implementation is contained in code that calls an object. Even the storage size for the object must be unknown or determined at run-time.

Using dynamic encapsulation, it is possible, for instance, to write and compile code that can input a series of items, store them in a list, and then sort the list, without any knowledge about the implementation of the individual items or the list. At run-time, the executable code is called with two objects: one that acts as a constructor for the items, and one that acts as a list. The only knowledge contained in the code is that the constructor object can be invoked to create a new item, the items can be compared to determine their order, and the list can be appended to and have items swapped. Given only this knowledge, dynamic encapsulation allows the rest of the implementation to change at run-time.

Dynamic encapsulation forms the foundation upon which the Dynamic Encapsulation Model is based. This model will be discussed in *Section 4 (Dynamic Encapsulation Model)*.

2

The Existing C++ Model

C++ attempts to provide much of the functionality of object-oriented programming languages together with the efficiency of C. As such, it has received criticism both from those who feel it is not a proper object-oriented language, and from those who feel its efficiency suffers under the weight of too many features. Throughout the history of computer programming languages, most popular languages have been criticized for various styles of programming that they do or do not allow, and certain features have been said to always lead to bad code, and thus they are supposedly ‘‘bad’’ features that should never be used. A good example of this is the `goto` statement, which is often the cause of much debate [*Dijkstra68*] although it can be shown to have its uses in certain situations. [*Kesteloot96*]

A variety of papers have already gone to great lengths to criticize C++, ⊕ so rather than taking a strictly critical approach, this chapter will attempt to present the various aspects of C++ in terms of the object theory previously described, so that no language features are ever banned, but instead shown to sometimes not represent the intended model; in this way it is hoped that a full understanding of all available features may be used to design a system, rather than just a simplified set of rules. Often, programming ‘‘rules’’ would have to be broken regardless, for the sake of optimization. Especially in the cases of such optimizations, it is important to understand the model represented by the code, and what assumptions are being made, so that the abstract model is valid for the results that are considered important.

2.1 Features of C++

2.1.1 *Simple Types*

The simple types in C++ are such datatypes as `int`, `char`, `float`, `double`, and their various sizes, such as `short` and `long`, as well as their `unsigned` counterparts. [*Stroustrup93*] All variables created with such types can be seen as containers for simple tokens that represent the abstract concepts of characters and numbers.

⊕ An excellent source of background material before reading this chapter would be [*Joyner96*]

It is important to recognize the limitations of each of these types. For instance, on a 32-bit architecture using twos-complement integer representations, the `int` type can only hold integer values from -2^{31} to $2^{31} - 1$. Therefore, any attempt to store a value outside this range into an `int` variable, or any attempt to store a non-integer value, will invalidate the computational system. In most cases, the compiler will allow such assignments and will not generate any run-time checks to ensure that the full representation of the token for the number can be stored in the container without truncating it. All such checking is left to the programmer, and numerical inaccuracies can easily result if these limitations are not taken into consideration.

The simple built-in datatypes usually can be seen as *types*, in the sense described in *Section 1.3.2 (Types)*, as long as a full understanding of their limited range is available. * However, each of these *types* is also bound to a specific internal *class*, which is architecture-dependent, and cannot be changed. As long as the program accesses an instance of a simple datatype as if it were an encapsulated object, the architecture-independent type is used. When the address of such an instance is taken and the memory image of the instance is examined or modified as something other than the entire simple datatype, then knowledge of the architecture-dependent class is used and the code is no longer guaranteed to be portable. Essentially, by taking the address and looking at the individual bytes, the token is being seen as an object and is being broken down into parts based on its implementation. This is the cause of many data incompatibilities, such as those between *big endian* and *little endian* machines, and handling these incompatibilities becomes even more difficult when dealing with varying floating point implementations.

2.1.2 *Pointers*

Pointers in C++ can be used both for referring to containers and for extending tokens. In essence, all pointers are tokens that act as references to other containers, but if only one instance of a given pointer exists, it can sometimes be seen semantically as being an extension, rather than a reference to a new container.

For example, in most linked lists, nodes of the list are not shared with other lists. Instead, although pointers are used, the entire list essentially acts like a single object. In this sense, the pointers are being used to extend the *token* that represents the *extensional* list, so that the list may be of an arbitrary size. Every time any node in the list changes, the list as a whole becomes a new token that represents a different extensional list. Even a pointer to the head of the list is still part of that single *extensional* token that represents the list, but the address of the head pointer is the token that represents the *intensional* list. Functions which require only the extensional list just need a copy of the head pointer, but functions that require the intentional container for the list need the address of the head pointer. Sometimes it is possible to cheat and pass a copy of the head pointer for updating the list if it is known that the first element already exists and will not be modified. In this case, it should be realized that a limited form of the intensional container for the list is being passed to the function.

It is actually possible for multiple pointers to point to the same location and still be acting as extensions of a single token rather than acting as a shared container. This is often done to conserve memory and is easy to implement if the shared area is read-only. If the shared extension

* Unfortunately, it is not always possible to know the limitations in a portable manner. For instance, there is still a dispute about the size of `long`s and `int`s for different architectures, such as 32 or 64-bit machines. However, by using a standard set of `typedef`s that allow specification of an integer based on its size or purpose, this problem is mostly alleviated.

might be modified, then some form copy-on-write (COW) semantics must be used to allocate a new region for changes so that the change does not affect all tokens that share this region. This might be combined with reference counting to prevent this copy when only one pointer to such a region exists.

It is important that the intended purpose of every pointer is known, so that they are used correctly to either create dynamically extendible tokens, or to represent references to other objects. Sometimes pointers directly map into the object model, and other times they are just a hidden part of the implementation. Either purpose is acceptable, but inconsistent use of a pointer can lead to bugs.

It has been claimed by some that pointers completely undermine the type system. [*Joyner96*] When combined with casts, which will be described later on in this section, this is obviously true. However, if pointers are used strictly for the type they are declared to point to, then they simply act as object references for in-memory objects and this should not violate any aspect of the type system. Pointer-arithmetic, on the other hand, can introduce problems. By allowing additions and subtractions to be performed directly on pointers, without bounds, it is possible to set pointers to arbitrary addresses which may not really contain objects of the specified type. Combining pointer arithmetic with a strong sense of array bounds could keep it type-safe, but C++ does not provide this.

2.1.3 *Const and Volatile*

A `const` qualifier can prefix the type description to specify that the object of that type cannot be modified. Used in conjunction with pointers and references, this can be used to specify read-only containers. There is no corresponding qualifier that can be used for write-only containers. †

A `volatile` qualifier can prefix a type to specify that the object of that type can change at any time. This means that all reads from the object must actually force a read and not use a previously cached value. It also means that all writes must actually be written back to the object. The `volatile` qualifier is generally used for hardware or portions of memory that are shared between multiple threads. The `const` and `volatile` qualifiers can be combined to specify objects that can cannot be locally modified but whose contents may be externally changed at any time. In some sense, `volatile` can be seen as a way of distinguishing intensional read-only objects from extensional ones.

2.1.4 *Classes*

Classes in C++ simultaneously share the role of *type* and *class*. They define a specific implementation, thus acting as *classes* (as defined in *Section 1.3.1 (Classes)*), but since there is no separate notion of a *type* mechanism, they must also fulfill this role as well. [*Martin93*] This is perhaps the biggest downfall of the standard object model presented by the C++ language.

This limitation must have been recognized by the creators of Java. The Java language inherits a class system similar to that of C++, but provides a separate concept of an *interface*, which is defined with the `interface` keyword. [*Flanagan96*] C++ lacks this feature, although abstract base classes can sometimes come close.

† Originally, C++ had both `readonly` and `writeonly` qualifiers. `readonly` became `const`, and `writeonly` disappeared. [*Stroustrup95*]

Through the use of virtual functions, which will be discussed in more detail in *Section 2.1.6 (Virtual Functions)*, *abstract base classes* can be created which act very much like *types*. Unfortunately, they can only be applied to the objects that originally inherited them; if new types are created in a system, then all objects that are semantically compatible and are intended to be used as objects of that new type must have the new type specifically added to their source code, and all such code must be recompiled.

In general, the rules of conformance in C++ allow substitution based on subclasses alone, using the class inheritance hierarchy to determine subclasses. This ensures correctness, at the expense of some flexibility, as long as entire extensional object representations are copied. However, when pointers or references to objects are used instead, essentially a token that refers to a container is being substituted, and the rules for substituting containers are *not* the same as those for the contents of the containers, as shown in *Section 1.6.5 (Container Substitutability)*. In fact, the type of the container must match exactly for conformance. C++ allows containers for subtypes to be substituted, thus breaking the rules of a safe type system. ‡ If a C++ pointer or reference is specified as `const`, then this problem does not exist, because this is equivalent to a read-only container, or a shorthand for directly copying the extensional representation of the contents.

Ian Joyner [*Joyner96*] says that the C++ model of encapsulation is broken, primarily because C++ equates encapsulation with data hiding, requiring the programmer to explicitly create and invoke accessor functions to provide independence from implementation details.

2.1.5 *Casts*

Casts provide a way of forcing an object to be a different type or class. They can be used to transform the data of an object or to simply see the existing data as being of a different type or class. There are four basic forms of casts in C++:

- `static_cast<newtype>(object)`
  This is used to convert an object of one class to another or to convert a pointer to an object of a different class. Conversion operators will be used where appropriate.

- `dynamic_cast<newtype>(object)`
  This provides a way to use a pointer to a base type and later convert it to a pointer to a derived type. A dynamic cast will check the run-time type of the object and return a null pointer if the object is not of the new type.

- `reinterpret_cast<newtype>(object)`
  This is similar to `static_cast` except that no data conversion is done. The token representing the object is just reinterpreted as being a representation of the new type.

- `const_cast<non-const type>(object)`
  This simply casts away `const` and does not actually change the type. Since `const` acts like a contract that says an object will not be changed, it seems odd that it should be necessary to break that contract in some cases to modify the object.

It is also possible to use "old-style" C casts, which will generally act as a `static_cast`, `reinterpret_cast`, or `const_cast`.

‡ If inheritance is purely structural, and the inherited components are completely independent from the new components, then this is acceptable. Inheritance is not always used in this way, however.

C++ relies heavily on casts to circumvent problems in the type system. This is a carry-over from its C-based ancestry and the unfortunate consequence of it is that it is frequently necessary to undermine the type system. [*Joyner96*]

The mere existence of casts makes it impossible to fully trust the type declarations of a function. Regardless of what is claimed, it is possible for the implementation to treat the objects as completely different types, thus creating implementation-specific dependencies. A more flexible model of types might allow the development of programs that did not need to circumvent the type system.

2.1.6 *Virtual Functions*

Virtual functions allow a derived class to have member functions that override the member functions in a base class. This allows a subtype to define a complete implementation and still be correctly subsumed to a supertype by ensuring that the correct functions are still called for the implementation of the object.

By making all base classes use virtual functions, classes can take on the role of type interfaces. In fact, any class whose public functions are all virtual can be seen as an interface and can be used as such. Pure virtual functions can also be defined to make an interface complete in the absence of an actual implementation. A pure virtual function is one that is initialized to 0 in the declaration. In C++, this means that no implementation will be given, and it prevents instantiation of objects from the class containing the pure virtual function. A class with one or more pure virtual functions is called an abstract base class and it functions very much like an abstract type with the limitation that it can only be inherited from and not directly instantiated.

2.1.7 *Templates*

Templates provide *genericity* by allowing classes and functions to be written in a generic parameterized way that can be customized by instantiating the template with a particular set of precise types and values. Templates add yet another level of complexity to the typing system (as well as the parser) in the interest of maximizing code reusability. They do not reduce the size of the object code, [¤] only the complexity of the source code; [◊] in many ways they act like type-safe macros.

There is an artificial dichotomy between class templates and classes. Generic instantiation from a template to a class can occur only once, and a generic instance cannot be used as a subclass. A better solution to the problem of genericity is presented in [*Palsberg94*] , which allows any class to essentially act as a template to form new ones, when proper substitution rules are used.

Due to their complexity in C++, templates will be beyond the scope of this thesis, although they may be used to augment some of the features discussed herein.

[¤] This is true of most current C++ compilers, but not necessarily true of the language itself. A C++ compiler is allowed to collapse identical code.

[◊] Although the templates themselves seem to add a lot of complexity to the source code, they can greatly simplify the code that uses them.

2.1.8 *Implementation Decisions*

C++ allows many low-level implementation decisions to be made directly by the programmer, such as whether a function is virtual or not, whether the extra indirection of a pointer to an object is necessary, etc. This allows fine-grained control over many implementation issues, but it also burdens the programmer with having to decide these optimization issues just to get a functioning program. It also means that major changes often have to be made throughout the code when one of these decisions changes, rather than letting the compiler do the work. There is too much emphasis on *how* things are to be computed, rather than *what* is to be computed. [*Joyner96*]

For instance, for maximum flexibility, all functions should be declared as `virtual`, but this results in sub-optimal code and excess baggage for every instance of the class if this feature is not necessary. This decision must often be made in advance when a base class is designed, before it is even known whether the derived classes need the function to be virtual. Once again, the interface and implementation in C++ are closely coupled, to the extent that a change in one necessitates a change in the other.

2.2 Types and Classes

It was previously stated that the merger of types and classes might be the single biggest flaw in the design of C++. Since this thesis focuses on a solution to that problem, this section will explore that issue further.

2.2.1 *Single Hierarchy*

In C++, a single hierarchy exists for inheritance, which combines both the interface and implementation for classes. This means that if an interface to a class is to be reused and extended by a derived class, the implementation must be inherited as well. It is possible to override some parts of the implementation using virtual functions, but the field structure of the base class is always inherited and can only be extended, not changed.

It is also not possible to inherit from an implementation without inheriting the interface as well. It is possible to override functions in the derived class, making some of them private, but this assumes that full knowledge of all functions in the base class is available [¥] and this is a cumbersome technique to control the interface to an object while reusing an implementation.

It is certainly possible to use an abstract base class with virtual functions as an interface, and then to have various implementations that inherit from it, and can be used in place of it, thus providing a separation of interface and implementation. However, this incurs the overhead of a virtual function table pointer for every instance of the class. If, for example, the objects are quite small and are being stored in an array, this could create a very large unnecessary overhead, especially since every object in the array is already known to have the same implementation.

[¥] It is possible to add new functions to the source code for the base class after the derived class was written, in which case those functions will now be exposed through the derived class as well, whether or not this was the intent.

2.2.2 *Substitutability*

The builtin subsumption rules in C++ will always allow the derived class (or a pointer to it) to be used in place of the base class (or a pointer to it). In essence this means that inheritance is used to only extend classes, not change them in other ways. The layout of the fields in a base class is strictly maintained in all derived classes, and the functionality defined for the base class must still be valid for the derived class. Once again, with virtual functions, it is possible to override the functions, however the fields cannot be changed, and the extra overhead of the virtual function table pointer is once again incurred for every instance of the class.

2.2.3 *Static Classes*

The notion of a class in C++ is a rather static concept. It defines simultaneously the layout of a data structure, the code for operating on that data structure, and the public interface to that code. Without using pointers to break the type system, there is very limited flexibility in the layout of a class [@] and no support for extending or changing the implementation of an object that already exists.

2.2.4 *Are Circles Ellipses?*

One example of the shortcomings of basing subtype relationships on a static class hierarchy is the conflict when attempting to model circles and ellipses. In mathematical terms, all circles are ellipses, but only certain ellipses are circles. Thus, circles can be seen as *subtypes* of ellipses. However, in C++, there is no easy and straightforward way to model this correctly.

Example 2-1 shows an initial attempt at modeling circles and ellipses in C++. Notice that the circle could be implemented either by adding a new member variable for the radius, or by just setting the major and minor axes, and deriving the radius from them. However, there are still problems with both of these approaches.

First of all, the most efficient class for implementing a circle would be one with just a radius. However, in order for the circle to be subsumed to an ellipse, it needs to be derived from `Ellipse`, and thus it inherits the member variables of an ellipse as well.

Secondly, and even more importantly, note that neither of these classes allow changing the circle or ellipse after it has been created. Certainly, it would be no problem to allow the radius of the circle to change. However, changing the major or minor axes of an ellipse would be disastrous if the ellipse was originally declared as a circle, since it might no longer be one, even though it was statically typed as `Circle1` or `Circle2`. This problem stems from the fact that circles are specialized forms of ellipses that limit the domains of the attributes. In other words, although the *radius* attribute appears to have been added, it is actually derived from the semi-major and semi-minor axes, and requires that both axes are equal, a restriction that the ellipse did not have.

These problems can be remedied somewhat by splitting the type interface and the implementation for ellipses and circles, as shown in *Example 2-2*. Technically, since C++ has a combined notion of type and class, this did not need to be done for the circle, unless specialized types

[@] The `union` construct can be used to make some parts of it flexible, to a limited degree and in a predefined manner. To some extent, however, this can already be seen as breaking the type system.

Example 2-1: C++ Circles and Ellipses, first attempt

```
class Ellipse {
protected:
    const int _major;
    const int _minor;
public:
    Ellipse(int major, int minor) : _major(major), _minor(minor) {}
    int getMajor() const { return _major; }
    int getMinor() const { return _minor; }
};

class Circle1 : Ellipse {
    const int _radius;
public:
    Circle1(int radius) : Ellipse(radius*2, radius*2), _radius(radius) {}
    int getRadius() const { return _radius; }
};

class Circle2 : Ellipse {
public:
    Circle2(int radius) : Ellipse(radius*2, radius*2) {}
    int getRadius() const { return _major/2; }
};
```

of circles will later be derived.

Now ellipses and circles can both be created in their most efficient form, and every circle can be viewed as an ellipse, but can only be changed as a circle. [#]

Unfortunately, this still does not allow all circular ellipses to be seen as circles. Some objects that are created as an `Ellipse` might have equal major and minor axes, in which case it should be permissible, with a run-time check, to view these as `TypeCircle` and to get the radius. In fact, to go one step further, it should be possible to then set the radius as well. C++ provides no easy means to accomplish this, although a first attempt might be to add a function to TypeEllipse that performs a test for circularity and returns either an object derived from TypeCircle, or a null pointer.

The problems here are all results of the way that inheritance in C++ is done via extending a base class. New fields and methods can be added, but none can be removed, and the substitutability rules are based solely on the class inheritance hierarchy. New methods can override existing ones, but only when they are invoked on objects known to be instances of the derived class. When the object is used as the base class, the original methods still get invoked. Through the use of virtual functions, the new methods can be invoked. However, there is still a problem with the fields; although new fields can be added, none can be removed or changed. All of these problems are the result of the *type* (or interface) hierarchy being the same as the `class` or (implementation) hierarchy.

In some cases, it makes sense to have the *type* and *class* hierarchies be identical. When new features are added to some object, new fields are needed to store the extra information, and the new derived object is truly just an extension of an existing one. However, with circles and

[#] Note that this is due to the difference between the abstract concept of a circle of a particular radius, and the container for a circle, which can potentially contain any arbitrary abstract circle. If the container were an ellipse container, then even if it contained a circle, it could be changed as an ellipse.

Example 2-2: C++ Circles and Ellipses, second attempt

```
class TypeEllipse {
public:
    virtual int getMajor() const = 0;
    virtual int getMinor() const = 0;
};

class Ellipse : public TypeEllipse {
    int _major;
    int _minor;
public:
    Ellipse(int major, int minor) : _major(major), _minor(minor) {}
    int getMajor() const { return _major; }
    int getMinor() const { return _minor; }
    void setMajor(int major) { _major = major; }
    void setMinor(int minor) { _minor = minor; }
};


class TypeCircle : TypeEllipse {
public:
    virtual int getRadius() const = 0;
    int getMajor() { return getRadius()*2; }
    int getMinor() { return getRadius()*2; }
};

class Circle : public TypeCircle {
    int _radius;
public:
    Circle(int radius) : _radius(radius) {}
    int getRadius() const { return _radius; }
    void setRadius(int radius) { _radius = radius; }
};
```

ellipses, the circle is a *specialization* of an ellipse, which requires the storage of less information, since its attributes are more constrained. Thus a proper implementation for storing a circle should not need to store both axes, since they are equal. Even if storage size is not a consideration, there is still the problem of maintaining the constraints. If some object is supposed to represent a circle, and it is modifiable, then the view of it as a modifiable ellipse cannot be allowed, since that would not enforce the constraint that the major and minor axes must stay identical. Such an object can be seen as a readable or writable/modifiable circle, but can only be seen as a readable ellipse. Any object that can be modified has to follow the subsumption rules for containers, as described in *Section 1.6.5 (Container Substitutability)*.

In *Section 4.7.1 (Circles Are Ellipses)*, it will be shown how a more flexible model can be used to correctly implement the relationship between circles and ellipses and allow them to be interchanged appropriately.

3

Extending C++

With the goal of adding support for dynamic objects to C++, some way of extending the C++ language had to be developed, either via new support libraries or through true syntax extensions. Over several years, many experimental object frameworks were implemented with varying degrees of success.

3.1 Previous Attempts

Many different techniques were investigated for the implementation of dynamic objects in C++. The earliest techniques employed a very different model of objects which was similar to the model in the original Shag/OS. Attributes and methods, collectively referred to as **features**, could be dynamically looked up for an object by knowing only the feature name.

Two major drawbacks existed with this system:

- A run-time search was needed to match an arbitrary feature selector [□] with its implementation. This could be optimized by passing an index or pointer back to the caller, which could be used as a *hint*, [⊕] but it still incurred a large run-time overhead.
- Ambiguities could exist with respect to the intended semantics for a method selector, especially with operators. For instance, the '+' operator could mean numeric addition when applied to numbers, or concatenation when applied to strings. The meaning might be clear when the implementation is known, but for dynamic objects the intended meaning could change.

Both of these problems stemmed from the fact that the original object model was *untyped*. Any object could be of any type, and although run-time checks could be performed to determine if a method with a specific name was supported by an object, there was no way to match the signatures of methods.

[□] A feature selector in this system was a 32-bit value that was uniquely associated with each identifier string used in the system.

[⊕] This hint was some piece of information that was maintained to improve performance, by providing a suggestion of a likely place to start searching. In the generic object implementation, the hint was just the index of the last invocation, so that repeated identical invocations would not require any searching unless the object's structure changed.

The implementations that followed attempted to address the type issue and also reduce the overhead required for binding a method selector to its implementation. Many of these implementations still tried to work entirely within the bounds of the existing C++ language, but they always seemed to either lack flexibility, or required many overly explicit declarations and much extra code that was simply an artifact of the object model instead of being an essential requirement of the algorithms.

3.2 Requirements

Although a multitude of useful ideas had come out of the many experiments, it became obvious that a set of requirements needed to be decided on, and something needed to be developed that could meet those requirements. This led to the separate development of a language independent object model, called the *Dynamic Encapsulation Model* (DEM), which will be introduced in *Section 4 (Dynamic Encapsulation Model)*. C++ could then be extended to work with dynamic objects. In order to support the Dynamic Encapsulation Model and still work reasonably well with existing C++ code, the following requirements were chosen, in addition to those imposed by the DEM:

- It should be possible to take an existing C++ class, and turn it into a dynamic class, without changing any of the existing code or header files.
- There should be as little overhead as possible when invoking a method, after the type interface is bound.
- Compiled code using only the type interfaces should have no dependencies on the class system used. This code should be able to utilize objects from different object and class systems simultaneously. This also means that recompilation should not be necessary to pass new objects with different implementations to previously compiled code.

3.3 DC++ and DECO

Some form of interface specification was clearly needed for the new dynamic interface, and since existing *Interface Definition Languages* (*IDLs*) did not meet the needs of this model, a new form of IDL had to be written. This new IDL shall be referred to in this document as DC++. It is based on the C++ language, with extensions for supporting dynamic classes and types. DECO (the Dynamic Encapsulator of C++ Objects) is the program that takes DC++ as input, and produces standard C++ as output. DECO can be used to either compile DC++ as an IDL, generating code and header files to link with other C++ code, or it can be used as a full compiler for the DC++ language.

Greater compatibility with future code implementations is gained by providing all of the dynamic functionality through new language extensions, rather than just through specialized libraries and header files that the user is required to include. If the support was provided as a library, then the code using it would have some dependencies on the implementation of object type and class interfaces. Creating new language extensions for types and classes as separate concepts eliminates those dependencies. Changes can be made to the implementation of the object system, with respect to how an object is stored, how methods are dynamically accessed, etc., without affecting programs that use the system, since the language extensions allow specification of the dynamic features in a higher level form, and DECO generates the implementation-specific details.

Of course, a fully dynamic object-oriented system could be implemented in pure C, without any C++, and certainly without DECO. This fact can easily be demonstrated by the fact that the first C++ compilers (such as CFront) translated C++ code into C code, and DECO translates DC++ code into C++ code. However, writing dynamic object-oriented code directly in C or C++ would unnecessarily burden the programmer with many details about the management of objects, and would require rewriting much code if changes were made to the underlying implementation. It would also make the code less portable, since many aspects of making an efficient dynamic interface are specific to certain architectures. DECO hides these details, allowing the programmer to write code that is only concerned with objects and their dynamic interfaces.

Language extensions to properly support a concept allow a programmer to be more concerned with *what* is to be done, rather than *how* it is to be done. An ideal system would enable a programmer entirely prototype the high-level ‘‘what’’, and then subdivide all those parts into ‘‘how’’, reusing existing implementations where appropriate, and creating new implementations by subdividing them until all parts are implemented. It should then be easy to change the implementation at any level, to optimize the system in different ways or adapt it to different environments, without changing the high level structure. By providing the ‘‘what’’ as a language extension, the actual concept is being directly applied and this allows the widest possible range of implementation changes while maintaining the original intent. DC++ attempts to provide the programmer with a way to define abstract types independently from their implementations and to write algorithms that use those abstract type interfaces along with classes that each implement one or more types.

4

Dynamic Encapsulation Model

This section describes the conceptual object framework that is used for DC++. DECO implements this framework, and details about implementation mechanisms will be described later; in this section, the focus is on the theoretical development of the object model.

4.1 Goals and Requirements

In designing a dynamic object model that could be used to extend C++, a number of different goals were considered and prioritized. The following list includes the major goals and considerations, starting with the most important:

- Code reusability should be maximized.
- Types and classes must be supported as separate hierarchies.
- The model should be independent of any particular programming language and should support many different languages.
- There should be no single way of dealing with class and object implementations, but instead the system should support a simple, low-overhead mechanism that allows a variety of object systems to coexist and communicate with each other.

  — Binding of method selectors to their implementations must occur at run-time.

  — Usage of an interface must not require any knowledge of the structure, size, or any other implementation details of the referenced object.

  — It should be possible to construct objects without knowing anything about their implementation at compile-time, not even their sizes.
- Size/speed tradeoffs should be easily supported, so that the object system can be geared either for low-memory consumption at the expense of speed or faster performance through the utilization of extra memory, simply by substituting some objects in the system, or even just tuning some parameters.

In addition, the following capabilities should be able to be supported by the model in the future, even if they are not currently fully supported:

- It should be possible possible to implement methods using machine-independent code that can be either interpreted or compiled on-the-fly when it is needed. This will allow objects to migrate completely to machines with different architectures.
- Machine-dependent code should be able to override the machine-independent code when it is applicable. This is to allow for architecture- or platform-specific optimizations.

4.2 Core Concepts

The basic concepts and entities of the Dynamic Encapsulation Model will now be presented. Keep in mind that these are meant to be free from any particular implementation, although a reasonably efficient implementation should exist for all of them.

4.2.1 *Objects*

Everything in the system can be used as an object, including types, classes, simple numbers, aggregates, and even references to other objects. The atomicity of an object is entirely up to its implementor. No matter how an object is structured in physical storage, interfaces can be made available to see it either as one single atomic object, or as a collection of smaller interacting objects. This decision must remain flexible, so that the implementation can change when the conceptual model of a system changes. In essence, an object is any entity in the system that can be uniquely referred to and accessed via an abstract interface.

Even functions can be objects, although they are unique forms of objects that define relationships between inputs, states, and outputs. Abstractly, a functional object is a mapping from the Cartesian product of all its inputs, including any implied inputs that come from state information, to the various corresponding outputs. Concrete implementations of functions are objects that consist of lists of instructions, and perhaps some information about the context in which those instructions are to be interpreted. *

An object is generally defined as having inputs, outputs, and some sort of state. Some objects, however, do not have a state that can be changed, † and therefore the state requirement needs to be relaxed. In fact, as was shown in *Section 1.4.3 (Containers)*, objects with state can be thought of as two separate objects-- one being an *extensional* object, and the other being an *intensional* object that acts as a *container* that holds a *reference* to the extensional object. This is purely a conceptual notion; the actual implementation may consist of gathering pieces of state from all over memory or even from external inputs in order to execute one of the methods. It is important to realize that an object represents a conceptual encapsulation; there is not necessarily a

* Note that *interpreted* refers to interpreting the correct meaning for the code, and is not meant to imply that the code cannot be native machine code; technically, even native code is eventually ''interpreted'' by the hardware.

† Any object with a constant state is equivalent to one without a state-- the outputs all become constant functions in this case. Compile-time recognition of a constant state can result in a significant performance optimization, since many run-time calculations can now be replaced with constant evaluations.

single contiguous region of memory corresponding to that object. It is also possible for multiple objects to share overlapping regions of memory.

Classes and types are specialized forms of objects that represent a different concept, and as such some people have argued that they should not be objects. However, by considering them to be objects as well (more specifically meta objects), the means of accessing remote objects, classes and types are all consistent, and obtaining information about objects can also be done in a consistent manner. In addition, hierarchies of hierarchies can be created, greatly increasing the expressive power of the system. These topics shall all be covered in more detail in the following sections.

4.2.2 *Types*

Types in the DEM are abstract representations of expected behavior. A type includes both syntactic and semantic elements. The syntactic part is the list of functions that the type must support. The semantic part is the implied behavior of those functions, which is not formally specified, but is associated with the name of the type, and must remain constant. Thus, when a type description is created, the semantics should be clearly documented, and once that type has been released outside of the realm of the original programmer, the syntax and semantics of the type must not be changed. New types can be created that extend the previous type, but the existing type definition has to be frozen once it is in use.

Types allow a separation of the concepts of implementation and interface. It is important that the functionality of a type be as narrow as is reasonable, to allow as many objects as possible to implement it. This means it is usually better to break a large complicated type into several smaller types that can be combined, provided that the smaller types could be useful by themselves. One important area where this needs to be done is the separation of functions that read and write a container object. Because substitutability rules for containers are the inverse of the rules for the contents (see *Section 1.6.5 (Container Substitutability)*), most containers would have very limited use if a single type defined all their characteristics. By instead creating a type for reading the contents of the container as well as a type for changing the container, it is possible to use a wide variety of implementations with those types.

The type for reading an extensional object would appear to be identical to the type for reading the extensional contents of an intensional container, and thus it would seem that only one type is needed for both of these cases. However, there is a subtle difference. If an object is known to be extensional, then every method of that object is guaranteed to return a constant when called with the same parameters. This is not true of objects which can only be read, since some other thread of execution may have write access and thus modify the object. This is similar to the distinction between normal variables and `volatile` ones in C and C++. Therefore, the type for reading an arbitrary object is a subtype of the type for reading an extensional object.

In many cases, another type for modifying objects is necessary, when it is not permissible or practical to entirely replace the object. This type allows the object to be changed based on its current state and the parameters of the methods, but does not by itself allow the object to be directly replaced or examined. Usually, a general type is then created which inherits from the types for reading, writing, and modifying an object, to allow full access to those who need it.

Splitting types into separate read and write components solves the problem of vertical modification with respect to class hierarchies. Normally, vertical modification in the class hierarchy cannot be allowed, which means that a subclass cannot restrict any of the existing attributes or

methods of the superclass. [*Chen96*] This is because the subclass also acts as a subtype in most systems, and functions for both reading and writing are contained in both the subclass and superclass. The circle and ellipse example is a case of vertical modification. Technically, the circle is an ellipse, but it restricts the properties of an ellipse. For purposes of reading only, it is possible to implement the circle as a subtype of ellipse, but for purposes of writing, the ellipse is really a subtype of the circle. By creating a separate type hierarchy, with types for reading and writing separated, the type hierarchies can be represented correctly (with the type hierarchy for writable types being an inverse of the readable one) and the classes for implementing circles and ellipses can each implement the appropriate types. The circle class implements types for a readable circle, a writable circle, and a readable ellipse. The ellipse class implements types for a readable ellipse, a writable ellipse, and a writable circle.

The ellipse can also support a dynamic coercion to a readable circle, which has a test for the ‘‘circleness’’ of the ellipse. This is a type conversion which cannot be evaluated at compile time, but which is instead tested with a dynamic bind at run-time. If the test fails, ‡ then the result will be a `nil` object, instead of one handled by the readable circle implementation.

4.2.2.1 *Conventions*

The following conventions should be used with types in the Dynamic Encapsulation Model:

- Operator overloading, within a given type hierarchy, should be for specialization only, not for completely different meanings.
- Overloading of function names, within a given type hierarchy, should be used only for specialization (in C++, this may not always be possible with constructors, due to the limited constructor syntax)

4.2.2.2 *Limitations*

All possible limitations for a given type should be taken into account when designing the type, and some means to detect these limitations should be made available. It is essential, for type safety, that a class conforms completely to the definition of any type it claims to implement. This means that the definition must have provisions made for any natural limitations that may occur.

For instance, although it may be possible to declare a completely generic abstract number type called *t_num*, with the associated basic arithmetic operations, it would be impossible for any concrete implementation class to ever live up to the expectations of handling an infinite range of numbers. Therefore, it is important that there is a mechanism that can be accessed via the type interface that allows programs to detect the numerical limitations of the underlying class. If a class called *D_Int*8, representing 8-bit signed twos-complement integers is used to implement *t_num*, then there need to be attributes that the program can query to find out that the class can only store integers (as opposed to all real numbers, or complex numbers), and that the range is limited to -128 to 127, inclusive. Ideally, there should also be some way to detect when an operation has overflowed, either as a return value, an exception, a separate flag, or a special ‘‘out of range’’ value. However, from the view of type safety, it is sufficient to simply make the limitations available as part of the interface and document them well, and leave it up to the programmer

‡ In this case, if the ellipse does not have equivalent major and minor axes.

to test the limitations to determine if an overflow is possible, and take measures to prevent it.

It should be clear by now that types should not be changed often, and only out of necessity. Therefore, they should be well thought out and aptly named. To minimize future churn, each type should have as few methods as possible, with a clear purpose for each one, and a simple concept that holds them all together. If a type needs extra optional functionality added, that functionality should be part of a subtype which inherits from the original type. All code should be written to use the most general types it possibly can, within the constraints of functionality and performance, in order to maximize the reusability of that code. A more general type has many more implementations that can be used for it.

4.2.3 *Classes*

Classes in the DEM are concrete representations of implementations. They may directly implement one or more types, which are clearly specified in terms of a mapping from the functions of each type to the correct internal implementation functions. Classes define the actual layout of objects in memory, whether that layout is contiguous or scattered, or even automatically generated on demand. Users of the type interfaces make absolutely no assumptions about the implementation of an object. It might not even have any storage at all. All of these details are encapsulated in the class, and all accesses go through the type interface.

4.2.4 *References*

All object manipulation is done through the use of object references, which are associated with a particular type. A reference is a *token* for some object, which may be *intensional* or *extensional*. References are bound to various objects, and can sometimes be rebound to new objects, thereby changing the token contained in the reference. References are not meant to be used as normal changeable variables, but merely as a way to keep track of the objects being used. If an iterator for a loop is needed, for instance, an intensional container object should be used, and a reference is then kept to the container. Re-binding a reference is considered to have a much higher overhead than simply changing the contents of an intensional object, so references are not meant to be used as general purpose variables, but rather as handles to variables, constants, and function arguments (which can generally be considered constant for the lifetime of the function invocation).

References are usually always bound to some object, but they may also be `nil`. This usually happens during conditional assignment because the object the reference is assigned to does not support the required type. In this respect, they are more like C++ pointers than C++ references, since C++ does not allow nil references. However, their method of initialization and the fact that they do not require explicit indirection make them act more like C++ references in all other respects.

4.2.5 *Managers*

Managers are objects whose sole purpose is to manage other collections of objects or data. The class of any object can be an object manager, in which case the local data for the object is really just some special tag the object manager uses to locate the rest of the representation.

Managers essentially act as proxies, either for remote objects, or objects whose representations cannot be directly acted upon, for a variety of reasons. The object representations might be compressed, or encrypted, or in some address space that is not directly accessible to the client

thread of execution. For whatever reason, it is not possible to just directly call some code with the object's data as the `this` pointer (to use C++ terminology), and therefore methods of the object manager are instead invoked, and handed a piece of data which tells how to locate, decrypt, uncompress, or perhaps create the intended object. The actual implementation of the proxy calls is up to the object manager, since it is responsible for creating an interface table when a client binds an interface to the object. This will often involve the generation of stub code for the calls.

4.3 Superobjects

Superobjects are dynamic parent objects. They can be used to provide *dynamic inheritance*. All requests to look up a type interface for an object are automatically forwarded to the object's superobject. Superobjects may be chained, but there must not be any loops, or infinite recursion could result during type binding.

Dynamic inheritance differs greatly in implementation from static inheritance, because any aspect of it could change at any time. During the time when a particular type is bound to an object, its superobject must not be changed, because this could result in dynamically changing the type of the object. This restriction could be relaxed, of course, if the bound type refers to the immediate object, rather than the superobject, but in practice this additional check is much more expensive to implement, in terms of run-time efficiency.

4.3.1 *Dynamic Multiple Inheritance*

Theoretically, it should be possible to support multiple superobjects, but due to the complications with ambiguity resolution, only one parent object is directly supported by this model. Dynamic multiple inheritance can be emulated by creating a superobject that contains references to the multiple superobjects, and which will invoke them in the correct order to resolve ambiguities. Note, however, that this will be a depth-first, rather than a breadth-first, search. If a breadth-first search is desired, each superobject will have to be checked for the existence of the requested type, and then each of the superobjects of the superobjects will need to be invoked, etc., until the desired type is found.

4.4 Forwarders

Forwarders are objects that can be used to intercept method invocations. They can be attached to an object, and serve to forward all calls to another object first, before invoking the calls on the original object. Since the original object is supplied to the forwarder as its superobject, the forwarder may selectively override individual methods of a type, by passing on the invocation of unsupported methods to its superobject, and hence to the original object that had the forwarder attached. Forwarders may supply new types of their own, as well as additional fields, but if the forwarder is detached, all such information will no longer be associated with the original object, and the object will appear to revert to its previous type.

The concept of forwarders was used in the mob programming language, [*Burdick89*] primarily as a way to implement magic. [*Riggs89*] The mob language was designed for writing interactive fiction, and as such, used object-oriented techniques for implementing the behaviors of real-world objects. In mob, a forwarder actually replaces the object it is attached to, and maintains a handle to the original object. The `Self` variable is essentially updated to point to the forwarder. [*Burdick90*]

Forwarders provide a means to selectively override parts of the normal behavior of an object, and later restore the object to its original behavior, allowing an object's behavior to be temporarily changed in useful ways that were not anticipated at the time the original object was created. Forwarders also allow a single piece of code to be applied to a variety of different objects, regardless of their implementation. In this sense, they are similar to a dynamic form of the *mixin* concept used to enhance objects in some other systems. [*Snyder87*]

For operating systems, as well as applications, forwarders provide a way to alter the behavior of an object in the system without requiring knowledge of the implementation of the object. For instance, command-line editing and history could be automatically added to all terminal I/O by adding a forwarder to the console I/O device object which performs such activity, while still utilizing the original console I/O object. Certainly, this could be done without forwarders if it was planned into the original system design. This would often be more efficient as well, but it is not always possible to design everything into a system in advance. The great advantage of forwarders is that they can be added into a system at run-time to enhance its behavior, even though this new behavior was not conceived of at the time the system was implemented.

In an operating system, forwarders could be used to dynamically extend the kernel, if it is considered to be an object, or a collection of objects. Virtual memory could be added, for instance, to a kernel that only supports contiguous memory, and all system calls involving memory allocation could be intercepted by a forwarder that provides the necessary extra functionality to support virtual memory. ¤

A forwarder could also be used to block access to specific types that an object supports, by simply acting as if they did not exist, or by passing them on to the superobject of the original object, thus bypassing the type implementation in the original object. This could be utilized, for instance, to skip over type implementations that might be known to have bugs, but for which the source code is not available. This could also be used to control security, as discussed further on.

In addition, a forwarder could be used to do profiling on all accesses to an object by simply recording the appropriate statistics, and then passing on the calls. It could also handle administrative accounting, by logging certain types of accesses, and perhaps investigating the caller of the object as well, before relaying the method calls.

A single forwarder may be attached to multiple objects, but each object may have only a single immediate forwarder. Forwarders may be chained, however, and multiple immediate forwarders may be emulated by creating a forwarder that keeps track of multiple object references, and selectively attempts to forward the call to each of them.

It should be noted that a forwarder often changes not only the implementation, but also the set of valid types for an object. As with dynamic inheritance through superobjects, it is important that a forwarder is not attached to an object while a type interface is bound to that object. Any bindings that occurred prior to attaching a forwarder will continue to refer to the original object. Although this appears to be a great limitation, it is necessary to ensure the validity of bound

¤ To handle issues of security, an object must not allow a forwarder to be attached to it unless the caller who is attempting to attach the forwarder has a security level greater than or equal to the creator of the original object. In an operating system, this generally means that kernel privileges are required to attach forwarders to any system objects.

types, and can be applied efficiently by binding the types for the smallest reasonable duration. Of course, overly short bindings can result in the high overhead of excessive bind calls, so there is a tradeoff here between flexibility and efficiency.

If a forwarder is attached while there are still bindings, most objects should defer the attachment of the forwarder, or fail the `forward` method call. It is possible, however, for the object to immediately implement the forwarder if all bindings were kept track of or went through a common interface. This could be done with an extra level of indirection, but it must be guaranteed to be valid so that the object is not left in an inconsistent state for different clients.

Although the original object appears as a superobject to a forwarder, the `superobject` attribute will not reflect this, since the goal of the forwarder is to act as a transparent extension of the functionality of the original object. Instead, the `original` attribute should be used when it is necessary to directly access an object that has a forwarder attached. This can also be used to check for the existence of a forwarder, although not to discover its identity.

The `forward` method may be invoked on any object to attach a forwarder to it. The forwarder may then be removed by calling the `unforward` method. Note that `unforward` will only remove the specified forwarder object, if such an object exists. This means that if multiple forwarders are attached in a chain, any one of them may be selectively removed, and the chain will be re-linked without that one. This also means that preventing the identity of a forwarder from being discovered is sufficient to prevent its removal, a useful feature for security reasons.

If an attempt is made to bind an object with a forwarder to an implementation class, then only the class of the forwarder is considered for the binding. If the class of the original object needs to be considered, then the `original` attribute must be explicitly used to obtain this object. Allowing direct binds to the original object, implicitly bypassing the forwarder, could violate security, and make the behavior of the forwarder inconsistent.

4.4.1 *Dynamic Optimization*

By attaching forwarders to classes, the system can be optimized *while it is running*. In other words, more optimal type implementations can be written, and simply linked into a class by using a forwarder. Calls to methods that are not yet optimized can be passed on to the original class. By adding the forwarder to the class instead of to a single object, it overrides the future behavior for all objects of that class. Note that any typed references that are already bound to objects of this class will remain bound to the original class, until such time as they are re-bound. When the entire functionality of the class has been replaced, the `replace` method can be invoked on the original class, to replace it with the forwarder. At such time, however, the forwarder is now just functioning as a normal class, and any references to `superclass` will simply refer to the original superclass.

4.4.2 *Security*

One of the most obvious uses for forwarders is for security. By attaching a forwarder to any object, all accesses to that object, for any purpose, can be intercepted. Such a security forwarder may not support the usual `forwarder` and `original` attributes, as these could allow the security mechanisms to be bypassed.

Many forms of security, such as access control lists, or simple read-write-execute permissions, can be implemented as various type of forwarders. In order to not violate this security, any object that supports forwarders must also keep track of a binding count. Attachment of a forwarder to an object with a non-zero binding count must fail, create a duplicate object, get deferred, or be suspended (blocked).

4.5 Persistence

Persistence of objects is the ability to save the state of an object to some sort of storage that persists longer than the lifetime of the program that created the object. This is made possible by giving objects a standard method that can be called to *flatten* them into a byte stream which can be saved just like any other contiguous chunk of memory. In order to ''reconstitute'' the object, it is necessary to know the class of the object so that a suitable implementation can be found or loaded and used to regenerate an in-memory form of the object. The details of this will be dealt with more fully in *Section 7.3.5 (Flattening and Packing)*.

There is also a *pack* method that can be called for shipping off objects to other address spaces where the same implementations might not even be available. The difference between flattening and packing is that flattening produces a byte-stream representation of the object that is associated with the *class*. This representation is tagged with a class signature that describes the layout of the flattened image. Although the exact same implementation does not have to be used to regenerate a function in-memory version of the object, any implementation that loads the flattened image must support the same layout.

In contrast, packed images are implementation-independent, and are associated with the *type*. As such, they can be sent off to remote machines and used to create objects which behave identically to the original. It should be noted that, with the exception of extensional objects, any packed or flattened object is a new object, and thus if it is unpacked multiple times, multiple objects are created, which are identical in contents at the moment of the unpacking, but which can then be changed independently. This is a form of duplicating objects, unless the original is destroyed and all references to it are redirected to the packed image or the resulting unpacked object.

Packed objects require machine-specific translation. For instance, at the simplest level, character formats and integer representations need to be changed to something uniform. Issues of big and little endian integers need to be dealt with in order to pack and unpack objects. All of this needs to be handled in the `pack` and `unpack` methods so that a byte stream can be sent to any machine and unpacked to an object that is identical in behavior. Flattening is constrained to work on machines with similar representations, and thus need not worry about these issues.

4.6 Remote Invocation

Remote method invocations involve a combination of the methods used for persistence, together with a means of data transport, and the ability to create proxy objects and proxy object managers for remote references. In addition, the signature of the method must be known, so that the individual arguments can be packed into byte streams for shipping off to the remote object.

For arguments that are intensional objects, it is usually necessary to create proxies that allow the method being called to access the original object. However, in some cases it may be

possible to pack a copy of the object, and ship it across the wire, with the knowledge that this is a duplicate acting as a cache, and is no longer a true reference to the original object. This is only valid when the object will be read, but not written, and when there are no synchronization issues caused by an interaction of other objects with this object in a manner which would change its state.

Arguments which are extensional objects can always be packed and shipped to the remote system, although there are cases where this may not be desired, such as when the object is unusually large. In such cases, the network latency needs to be taken into account and compared with the time it would take to just send the whole representation, rather than accessing pieces of it remotely through a proxy. This is an implementation consideration, and will not be discussed further in this section.

4.7 Examples

4.7.1 *Circles Are Ellipses*

By separating the concepts of *type* and *class*, it is now possible to create types for circles and ellipses that are based on the mathematical classification that a circle is an ellipse. It is also possible to create implementation classes for circles and ellipses, with each one containing only the necessary fields to store the information required to uniquely identify the object.

To create types that ensure correctness, it is first necessary to separate the concepts of an *extensional* object from an *intensional* one. When dealing with extensional objects, no modification of properties is allowed, and so the types *x_circle* and *x_ellipse* can be created as follows: ◊

x_circle:
  `radius`: $\rightarrow$ *num*

x_ellipse:
  `major`: $\rightarrow$ *num*
  `minor`: $\rightarrow$ *num*

If intensional objects are to be created, then types should also be defined for reading the current extensional value of the objects. In this case, the extensional objects are really degenerate cases of the intensional ones, and so the intensional ones ought to be declared first, and the extensional ones can inherit their methods, instead of declaring the extensional ones explicitly as was done above.

◊ For simplicity, the positions of the circles and ellipses have been left out of these examples.

r_circle:
  `radius`: $\rightarrow$ *num*

r_ellipse:
  `major`: $\rightarrow$ *num*
  `minor`: $\rightarrow$ *num*

x_circle: *r_circle*
x_ellipse: *r_ellipse*

The only additional property here is the implication that clients using the *r_circle* and *r_ellipse* types cannot assume that the value is the same every time the methods are invoked.

Next, types for modifying the intensional objects, or containers, need to be defined:

m_circle:
  `radius`: *num* $\rightarrow$ *statechange*

m_ellipse:
  `major`: *num* $\rightarrow$ *statechange*
  `minor`: *num* $\rightarrow$ *statechange*

These are essentially equivalent to creating a new object based on a function of the old object and the supplied parameter, but are usually implemented by just changing a piece of the state of the object. In some cases, however, it may be necessary to completely rewrite the representation of the object in order to execute a modification that only appears to change one portion of the total state.

4.7.2 *Removing Finite Restrictions*

In most functional languages, variables are declared to be of a particular datatype, with the datatype restricting the range of allowable values. For instance, if variable *x* is of type `short int`, it is limited on most 32-bit architectures to a 16-bit quantity with integer values in the range of -32768 to 32767, inclusive. Every function requires its parameters to have datatypes. This limits the range of values upon which the function may operate, and means that a general purpose algorithm cannot be coded up once and later applied to data of different types and sizes, unless it is written as a macro or a template. Even in such cases, however, the code is generated multiple times, once for each datatype, causing redundancy in the system. [¥] Also, the original source code must be recompiled if new datatypes are added.

With types as a concept separated from the implementation, it is now possible to create a type, which shall be called *num* here for the sake of example, that represents all possible numbers. Note that this type is not limited by any of the standard finite limitations of computers. Of course, it is not possible to create a container that can truly be said to be of type *num*, due to the fact that all containers in the computer's memory are limited to a finite amount of storage, but type *num* as a concept is infinite, since it can made as large as the available storage.

[¥] Sometimes this is desirable, when speed is far more important than size.

Functions may be written that take arguments of type *num* and return such types as well. In this way, algorithms can now be coded directly into functions and the actual code, at run-time, can adapt to whatever real data it is given. Note that limitations still exist; for instance, the function cannot create local variables of type *num*, except by cloning its arguments. [@] For that matter, it is not possible for any code to ever create variables that are simply of type *num*, since the type is merely a way of accessing a class. By cloning arguments, or by extracting their classes and then creating new objects of the same classes, it is possible to create new storage containers that have as much accuracy as the data being passed into the function. The function still cannot deal with infinite numbers, but it can deal with the largest data that can be passed to it, [#] rather than being arbitrarily limited by the datatypes chosen when the function was coded.

[@] It is actually possible for the code to create objects of type *num* if the code has knowledge of some classes that implement type *num*. In such cases, however, it is not guaranteed that the implementations chosen will be able to handle all the values needed to be compatible with the objects of type *num* that were passed to the code. Having the code depend on any concrete implementation places fixed restrictions on the code. Cloning the arguments creates new objects without putting such restrictions into the code. Another way of keeping the code free of implementation details would be to explicitly pass constructor objects as arguments. The code could then invoke these constructors to create new objects with the implementation chosen by the caller.

[#] This assumes that the results still fit in the same size containers. Properly dealing with numerical overflow is a complex issue, and is beyond the scope of this thesis. For some thoughts on the matter, see [*Kent94*]

5

The DC++ Programming Language

The DC++ Programming Language is essentially standard C++ with extensions added to support abstract types and dynamic classes. In its current implementation, templates and exceptions are not handled directly in DC++, but code that uses these features may still be linked with DC++ code.

5.1 Naming Conventions

Although no particular naming convention is enforced by DECO, the following conventions have been used during its development, and are used in all of the examples:

`r_typename`
: read-only types for intensional objects. Although this interface is read-only, the referenced object might still change from one read to the next, if another thread has write access to it. (the object is essentially "`const volatile`")

`x_typename`
: extensional object types. Generally, `x_typename` will inherit its signature from `r_typename`, with the implied restriction that the object referenced can never be changed. Accesses via `x_typename` are value-based instead of reference-based.

`w_typename`
: write-only types (for writing to intensional objects-- all operations must completely change the object and not depend on existing information)

`m_typename`
: modifiable/mutable types (for updating intensional objects-- the current object must be guaranteed to be of the corresponding `r_typename` type, since some of its information will be used). Often, this type will not exist separately, but instead this functionality will just be defined as part of `t_typename`, unless there is a reason to want the ability to update an object without being able to read it or change it to a completely new object. This ability might be useful for objects that handle logging or security.

`t_typename`
: full interface definition for a type, generally a read/write/modify type. This is for intensional objects currently containing extensional data of the same type, and is is generally simply a derived class from `r_typename`, `w_typename` and `m_typename`. If

`m_typename` does not exist as a separate type, then functionality for modifying the object can be defined as part of `t_typename`.

`D_ClassName`
: dynamic classes

`d_ClassName`
: the object representing a dynamic class

`c_typename`
: the type for a constructor object for `t_typename`. This is an object that has one or more constructor functions that return a new object of type `t_typename`.

typename_
: a constant reference to an object of type typename. This is simply a shorthand for `const` typename `&`, and is used commonly as an optimization for parameters and return values, without affecting the semantics (as long as nothing can change the value while it is being used). Ideally, a more advanced compiler could automatically use references instead of copying whenever it is appropriate.

type `attrib() const`
: a function for getting the value of attribute `attrib`

`void attrib(`type`)`
: a function for setting the value of attribute `attrib`. Function overloading makes it possible to have several different functions that take different types of parameters for setting a single attribute, when it is appropriate to do so.

`_internal`
: names used internally for a class. Exported names generally should not start with an underscore, unless they are implementation specific or mainly used internally.

5.2 Keywords

To avoid causing too many lexical clashes with existing names in C++ language, a minimalist approach was taken to adding new keywords. The complete list of new keywords is as follows:

`dynamic`
: This is the only new keyword added to the language. When placed after the `class` keyword, it specifies that the class is a *dynamic* class, which will be defined in more detail below. Combined with the keyword `virtual`, it can be used to specify dynamic abstract types. It can also be used in front of a type specifier in a dynamic class or type to show that the match to that type must be determined at run-time, when binding.

5.3 Operators

One new operator was added to the DC++ language:

`?=`
: This operator is used for conditional assignment, which is a form of assignment that could potentially fail due to a type incompatibility at run-time. When used for assignment, the return value from invoking the operator can be checked for success. When used for initialization or re-binding of an object, the object itself can be checked for success.

5.4 Typedefs

The standard header file for DC++ defines the following types using C++ `typedef`s:

`byte`
: a single byte; used instead of `unsigned char` to more clearly state the intent when used for non-character data.

`sz*`
: used for a C-style zero-terminated string. `sz` is declared with a `typedef` to be equivalent to `char`, and `sz*` and `sz[]` are used instead of `char*` and `char[]` to clearly distinguish zero-terminated strings from pointers to single characters and arbitrary arrays of characters.

`csz*`
: a shortcut for `const sz*`, since it is used quite frequently. Note that a variable of type `csz*` can still have a new string assigned to it. The only restriction is that the string itself cannot have individual characters modified. Thus, `csz*` can be thought of as the type for an *extensional* string, while `sz*` can be thought of as the type for an *intensional* string, subject to whatever buffer size limitations might apply.

5.5 Macros

In the standard header file for DC++, the following macros are defined:

`type`
: This is a shorthand for `class virtual`, □ which is used to specify an abstract type.

□ Originally, this was "`virtual class`", but too many grammar ambiguities resulted from this. Likewise, "`class dynamic`" used to be "`dynamic class`." Although the latter seems much clearer, it ends up being far too ambiguous in the grammar, so eventually the decision was made that when a `virtual` or `dynamic` keyword is used with a class, it must come after the `class` keyword.

`dtype`
: This is a shorthand for `class dynamic virtual`, which is used to specify a dynamic type. Dynamic types were part of the original design, but have been eliminated, so this is now reserved for future use.

`dclass`
: This is a shorthand for `class dynamic`.

`CSTR(`*"constant string"*`)`
: This is a macro that converts a constant string into a datatype known as `CStr`, which has both an address and a length. This way, there is no overhead for calling `strlen()` to find the length of the string, since `sizeof` can find it at compile-time for static constant strings. It would be preferable to do this automatically with templates, but there appears to be no way to do so in C++. With future versions of DC++, it should be possible for the compiler to automatically convert constant strings to a `CStr` structure when it is appropriate to do so, eliminating the need for the `CSTR` macro.

: Any place where CSTR is used, the code would still function with just the string, and no macro, but would have the added overhead, both in size and execution speed, of having to dynamically determine the length of the static string. Therefore, it is best to use this macro whenever possible.

`nil`
: This is just an alternative name for NULL, in lowercase since it acts more like a language feature than the traditional user-defined macro.

`DBIND(`*t*`, `*r*`, `*o*`)`

`DCBIND(`*t*`, `*r*`, `*o*`)`

`DREBIND(`*r*`, `*o*`)`

`DCREBIND(`*r*`, `*o*`)`

`DASSIGN(`*r*`, `*o*`)`

`DCASSIGN(`*r*`, `*o*`)`
: These macros are provided for allowing C++ to interface with DC++, and are described in *Section 6.4.1 (Interfacing C++ with DC++)*.

5.6 Reserved Names

The following names get used by DC++ for special purposes, so although they are not actually keywords, they should be considered as such in the contexts in which they are used:

`dataclass`
: Inside of a dynamic class, `dataclass` is a class declared to represent the static class used for storing the actual representation of the data.

5.7 Predefined Names

The following types, classes, and objects must be predefined in the standard DC++ header file:

`t_any`
: This is the most basic built-in type interface, and can be used to reference an instance of any class. All other types are implicitly derived from it, and the only method it supports is `bind`.

`t_type`
: the basic interface to a type (interface description) object.

`t_class`
: the basic interface to a class (implementation) object.

`D_Type`
: the standard type implementation

`D_Class`
: the standard class implementation

5.8 Syntax

The syntax of the DC++ language is very similar to an extended form of the C++ language, but currently without any of the following C++ features:

- RTTI - RunTime Type Identification ⊕
- Templates
- Exception Handling

All of the syntax associated with these features has been temporarily disabled, although the keywords are still recognized by the lexical analyzer, and the parser contains much of the necessary code (commented out). Hopefully, future versions of DC++ will support these three missing features and integrate them well with the dynamic extensions.

⊕ DC++ certainly has a form of run-time type identification, since it is possible to conditionally assign objects to type references, but it does not currently support standard C++ RTTI.

The extensions to C++ for abstract types and dynamic classes are modeled on the syntax for declaring standard C++ classes. Only a few minor syntactical differences exist between dynamic classes/types and standard static classes. They are as follows:

- the new keyword `dynamic` follows the `class` keyword for both dynamic classes and types.
- the keyword `virtual` is used to distinguish a type from a class, by implying that everything in this definition is virtual, and hence is just part of an interface.
- where access specifiers (`public`, `private`, and `protected`) were previously allowed, type specifiers can now be given to declare the interface specified for the following members.
- initializers for methods can be set to the names of other methods, thus allowing method aliases.

The only other new syntax is the ability to conditionally bind and assign using the `?=` operator, which may be used for initialization as well.

5.8.1 *Types*

An abstract type is specified in DC++ by '`class virtual`' or just '`type`' using the macro from the *dcobj* header file. Dynamic types are defined with the following general layout:

```
class virtual t_type {
    rtype  func1(ptype1,  ptype2,  ...);
    rtype  func2(ptype1,  ptype2,  ...);
    ...
};
```

Essentially, the abstract type declaration is like a C++ class that only defines member functions. An example might be:

```
type t_switch {
    int is_on();
    void turn_on();
    void turn_off();
};
```

Types can also inherit from other types, using the standard C++ inheritance syntax. For instance:

```
type t_fader : t_switch {
    int value();
    void value(int);
};
```

In addition to simply defining an interface, types can provide default code that gets used if an object's implementation does not provide any. This code must work entirely by using other functions from the type, thus making no assumptions about the implementation. Some examples might be:

```
type t_circle {
    int radius();
    void radius(int);
    int area() { return 314159*radius()*radius()/100000; }
};

type t_ellipse {
    int major();
    int minor();
    void major(int);
    void minor(int);
    int area() { return 314159*()*major()*minor()/400000; }
};
```

It is also possible for a type to say that it implements another type. If type *A* claims to implement type *B*, then an object of type *A* can be bound to a type interface of type *B* even if the implementation of *A* knew nothing about *B*. The syntax used is similar to that used for protections in C++, except that instead of an access specifier, (such as `public`, `private`, or `protected`) a type specifier is used, in this case `t_ellipse`. All functions within this section must be defined, either in the `type` definition, or in some code that is linked to the final executable. Therefore the full syntax for specifying types with implementations follows this general form:

```
class virtual t_type {
    rtype func1(ptype1, ptype2, ...);
    rtype func2(ptype1, ptype2, ...);
    ...
t_type1:
    rtype func1(ptype1, ptype2, ...);
    rtype func2(ptype1, ptype2, ...);
    ...
t_type2:
    rtype func1(ptype1, ptype2, ...);
    rtype func2(ptype1, ptype2, ...);
    ...
};
```

An example of a type that implements another type could be:

```
type t_circle {
    int radius();
    void radius(int);
    int area() { return 314159*radius()*radius()/100000; }
t_ellipse:
    int major() { return radius()*2; }
    int minor() { return radius()*2; }
    void major(int n) { radius(n/2); }
    void major(int n) { radius(n/2); }
};
```


Two things should be clearly noted here. First, the `area` function is not explicitly implemented for type `t_ellipse`,* and thus the default implementation from `t_ellipse` is used. If there was no default, this would be an error. Second, there is a mistake in this implementation that cannot be avoided with the types as they were defined. Setting the radius to half the major or minor access may be okay if those two axes are always set equal (meaning that the ellipse is really being treated as a circle), but it is incorrect otherwise. This problem is yet another manifestation of mixing up containers and their contents. Both of these types are really container implementations that have expectations both about what is in them and what can be put in them. It is incorrect to say that a container for a circle can be seen as a container for an ellipse, although a container for a circle can be seen as having an ellipse in it. In other words, a circle container, such as `t_circle` could implement a read-only ellipse type, such as `r_ellipse` that does not have any functions for changing it. In many cases, to fully utilize the type system it will be necessary to break down types into their readable and writable components.

A full breakdown of the types for circle and ellipse containers would be as follows:

```
type r_ellipse;

type r_circle {
    int radius();
    int area() { return 314159*radius()*radius()/100000; }
r_ellipse:
    int major() { return radius()*2; }
    int minor() { return radius()*2; }
};

type x_circle : r_circle;

type w_circle {
    void radius(int);
};

type t_circle : r_circle, w_circle;
```

* The implementation of `area` for `t_circle` does not automatically get used as the implementation for the `t_ellipse` section. To incorporate it, the following would have to be specified:

```
int area() = area;
```

Note that the scope for the target of function aliases is the same as that for implementations-- just the main scope of the type being defined, not the implementation section.

```
type r_ellipse {
    int major();
    int minor();
};

type x_ellipse : r_ellipse;

type w_ellipse {
    void axes(int major, int minor);
w_circle:
    void radius(int r) { axes(r, r); }
};

type m_ellipse {
    void major(int n);
    void minor(int n);
};

type t_ellipse : r_ellipse, w_ellipse, m_ellipse;
```

It is not always necessary to break down types to this degree, but since types should not be changed once they are in use, it is usually a good idea to do so, to guarantee the greatest amount of flexibility in the future.

5.8.2 *Implementations*

Any aggregate, including classes, structures, unions, and types, may specify one or more types that are implemented by it, as follows:

```
Class_Header {
    ...
t_type1:
    rtype func1(ptype1, ptype2, ...);
    rtype func2(ptype1, ptype2, ...);
    ...
t_type2:
    rtype func1(ptype1, ptype2, ...);
    rtype func2(ptype1, ptype2, ...);
    ...
};
```

Type implementations are specified in the same manner as access specifiers, but the implementation for each type is seen as a separate scope. As such, names can be repeated without any conflicts. The actual scope that needs to be used, if a function is to be defined outside of the dynamic class definition, is *classname*::*typename*.

Functions can be included inline, defined elsewhere, or can refer to another function with a matching signature. The standard initializer notation can be used in an extended form as follows:

```
rtype func(ptype1, ptype2, ...) = D_Class::another_func;
```

This allows renaming a function without creating an unnecessary intermediate function. This extended syntax is not limited to use in classes-- in DC++, any function can be aliased by simply

assigning it to the name of another function which has a compatible signature.

Any functions of a type that are not specified in an implementation will use the defaults for that type. If no default function exists, then an error will be reported at link time.

There is also an additional form of implementation used when access to information in the type reference becomes necessary. This involves explicitly declaring the `this` parameter, and giving it a type name for its implementation, as follows:

```
rtype func(TRef *this, ptype1, ptype2, ...)
```

If a completely different implementation is used, it should have a different type name for the internal structure, so that any incompatibilities should get quickly caught be the compiler. It should be noted that the `TRef` used in the example above is not guaranteed to work with all version of DC++, as it is based on the implementation used by DECO.

5.8.3 *Dynamic Types*

A dynamic type is specified in DC++ by '`class dynamic virtual`' or just '`dtype`' using the macro from the *dcobj* header file. Dynamic types follow all the rules for standard types, and have the following general layout, just like types:

```
class dynamic virtual t_dtype {
    rtype func1(ptype1, ptype2, ...);
    rtype func2(ptype1, ptype2, ...);
    ...
};
```

The main difference is that dynamic types also generate a dynamic object with an object descriptor. This can be accessed as `t_dtype::dobj`. Its purpose is to provide descriptions of all of the methods available via that type. `t_dtype::dobj` implements `r_type`.

5.8.4 *Dynamic Classes*

A dynamic class is specified in DC++ by '`class dynamic`' or just '`dclass`' using the macro from the *dcobj* header file. Dynamic classes are then created as follows:

```
class dynamic D_Class : BaseType {
    // additional local data
t_type1:
    rtype func1(ptype1, ptype2, ...);
    rtype func2(ptype1, ptype2, ...);
    ...
t_type2:
    rtype func1(ptype1, ptype2, ...);
    rtype func2(ptype1, ptype2, ...);
    ...
};
```

Some dynamic class declarations:

```
class dynamic D_testclass : TestClass
{
private:
    int private_function();
public:
    int public_but_not_dynamic();
t_test:
    int some_function(t_sometype, int) = TestClass::some_function;
t_wbuf:
    int write(char *b, int len) { return ::write(b, len); }
};
```

Multiple base classes may be inherited, and additional fields may be added. Base classes may be static or dynamic classes, but it is also possible to create a dynamic class that does not inherit from any other classes. All `public`, `private`, and `protected` fields will be inherited from the base classes, and used in the creation of `dataclass`, which will represent the static form of the new compound datatype.

5.8.5 *Conditional Binding*

The conditional assignment operator, '`?=`' can be used for conditional binding, both as an initializer, and to later re-bind a `TRef` to a new object. In both cases, the `TRef` should never be used without first checking that it is non-`nil` since a conditional binding can fail if that type is not supported by the implementation of the object. Here are some examples of different uses of conditional binding:

- A basic example of conditional binding:

```
int
cvt(t_any a, t_any b)
{
    t_string s ?= a;
    ...
```

- Conditional binding combined with a check for success:

```
    if(t_file f ?= b) {
        ...
    }
```

- Conditional re-binding:

```
    &s ?= b;
    ...
}
```

Note that to re-bind, the syntax involves taking the address of the `TRef`. This may seem odd at first, but the rationale is that this is the only operation that changes the `TRef` itself. Normal assignment operates on the object referenced by the `TRef`. By taking the address, it is possible to access the reference itself and change it. This is provided purely as a convenience for situations like loops and complex conditionals that need to rebind a reference to different objects before

dropping through to common code. It is not advisable to try to discern the datatype of the address of a `TRef` and use it as a separate entity in the code. The address of the `TRef` should only be taken as part of a conditional assignment. This means that the syntax shown above is the only correct form.

5.9 Standard Headers

All DC++ programs implicitly include the standard header, as if the following line had been the first line of the file:

```
#include <dchdr>
```

This file contains the standard macros and classes necessary to implement abstract types and bind them to classes. This file may have many compiler-specific dependencies since it is automatically included by the compiler.

In order to use dynamic classes, an object representation model must be included. The standard DC++ model is used by explicitly including its header as follows:

```
#include <dcobj>
```

This file contains the standard object representation as well as classes and types for dynamic classes and types, making the system self-descriptive.

In addition, there may be standard type interfaces that ought to be included. The current standard interfaces and headers are listed in the appendices.

5.10 Examples

5.10.1 *String*

Example 5-1 shows a simple string class, which has been extended to make its functions available dynamically. At first, it seems to have excessive verbosity and appears to be declaring the same functions three times. However, it is important to note that the separation of classes and types requires that every type has a complete specification, showing specifically which functions are supported by all classes of that type. If more than one string class were implemented, only one type would be created, and would be shared by both implementations. In this sense, it is equivalent to a C++ abstract class, which would use virtual functions to provide similar functionality.

If the original string class did not already exist, and did not need to be used separately as a static C++ class, there would be no need to declare `String` and `D_String` separately. Instead, `D_String` could include the definitions of all the functions, as shown in *Example 5-2*, which assumes that the type `t_str` has already been declared. The static class can then be accessed as `D_String::dataclass`.

Example 5-1: `String1.dc`

```
// original pre-existing static class for dynamic strings
class String {
private:
    char *_ptr;
    size_t _len;
public:
    String() : _ptr(0), _len(0) {}
    String(csz *str);
    String(CStr &str);

    String& append(csz *str);
    String& append(CStr &str);
    String& append(String&);
    csz* data() const { return _ptr; }
    size_t length() const { return _len; }
};

// a type for strings:
typedef const type t_str &t_str_;
type t_str {
    t_str_ append(csz *str);
    t_str_ append(CStr &str);
    t_str_ append(t_str_);
    csz* data() const;
    size_t length() const;
};

// a dynamic class for strings:
class D_String : public String {
t_str:
    t_str_ append(csz *str) = D_String::append;
    t_str_ append(CStr &str) = D_String::append;
    t_str_ append(t_str_) = D_String::append;
    csz* data() const = D_String::data;
    size_t length() const = D_String::length;
};
```

In both of these examples, there is an explicit assignment in `D_String` for all functions that implement type `t_str`. This may seem unnecessary, considering that the names are the same in this example. However, this is a requirement because without this explicit assignment, it would be more difficult to determine whether a missing function should cause a default implementation from the type to be used, or whether a function from an inherited class that happened to have the same name should be used. Likewise, an extra function added to an inherited class could end up overriding a default, changing the intended behavior of the dynamic class. To avoid these problems, it was decided that the location of the function should be explicitly stated, so that it only uses the default from the type declaration if nothing is specified. If no default is specified and no specific implementation is given, a link error will result.

Even more verbosity can be removed by moving the function declarations out of the main part of `D_String` and into the `t_str` section, as shown in *Example 5-3*. As before, these functions can still be accessed by referring to `D_String::dataclass` to get to the static class, and referring to the functions as typename::function. For instance, `append` will now be called `t_str::append`, and can be accessed as `D_String::dataclass::t_str::append`. Generally, if a function is meant to be accessed using the static data, it should be declared

Example 5-2: `String2.dc`

```
// an implementation class for strings, fully self-contained
class D_String {
private:
    char *_ptr;
    size_t _len;
public:
    D_String() : _ptr(0), _len(0) {}
    D_String(sz *str);
    D_String(CStr &str);

    t_str_ append(csz *str);
    t_str_ append(CStr &str);
    t_str_ append(t_str_);
    csz* data() const { return _ptr; }
    size_t length() const { return _len; }
t_str:
    t_str_ append(csz *str) = D_String::append;
    t_str_ append(CStr &str) = D_String::append;
    t_str_ append(t_str_) = D_String::append;
    csz* data() const = D_String::data;
    size_t length() const = D_String::length;
};
```

Example 5-3: `String3.dc`

```
// an implementation class for strings, fully self-contained, and shorter
class D_String {
private:
    char *_ptr;
    size_t _len;
public:
    D_String() : _ptr(0), _len(0) {}
    D_String(csz *str);
    D_String(CStr &str);
t_str:
    t_str_ append(csz *str);
    t_str_ append(CStr &str);
    t_str_ append(t_str_);
    csz* data() const { return _ptr; }
    size_t length() const { return _len; }
};
```

separately outside of any particular type, and then type-specific functions should call it, or alias it.

However, in this particular example, it will be necessary to know the full names of the functions regardless, since they will be needed to provide definitions. One of the `append` functions could be defined as follows:

```
t_str_
D_String::dataclass::t_str::append(csz *str)
{
    return t_str::append(CStr(str, strlen(str)));
}
```

Be aware that any linkages with normal C++ code will use different names. These will be described in *Section 6.4.1 (Interfacing C++ with DC++)*.

After declaring t_String and D_String, dynamic string objects can be created with any of the following declarations:

```
String s1(CSTR("this is "));    // Example String1.C only
D_String ds1(s1);        // new copy of String
D_String ds2(&s1);       // shared copy of String
D_String ds3("Hello ");
D_String ds4(CSTR("Goodbye"));  // more efficient way
```

The CSTR macro is not strictly necessary in these examples. For that reason some examples have been shown with it, and some without. Its purpose is to find, at compile-time, the address and length of a string, so that a run-time call to strlen() is not necessary for strings whose size is already known. Obviously, this can only be used on constant strings, and it will thus fail on anything else. From this point on, CSTR will not be used in most examples, to keep them shorter, but in real code, its use will result in greater run-time efficiency and some reduction in code size. Hopefully, future versions of DC++ will eliminate the need to explicitly do this.

Note that all of the constructors defined for the inherited class carry over automatically to the dynamic class, but only if there is just one base class and no new constructors or fields are added in the derived class. This makes constructors follow the same inheritance rules as functions, whenever it is appropriate to do so. This ability to implicitly access constructors of a base class occurs for all types of aggregates (structures, classes and unions), not just dynamic classes, and allows for easily extending the functionality of a class without having to redefine all of the constructors. †

Static invocations can be performed directly on the strings: ‡

```
ds1.append("a string. ");
ds3.append("World");
printf("%s\n", ds2.data());
```

Or the strings can be bound to abstract types, using a dynamic type interface derived from an Object Call Frame:

```
t_str ts1 = ds1;    // both of these styles
t_str ts2(ds2);     // are functionally equivalent
```

Then, calls can be made using the type interface, instead of the known class:

```
ts1.append("More stuff is being added to this string");
printf("%d\n", ts1.length());
```

When using bound types as parameters, implicit bindings will occur, and new bound type

† It is unfortunate that the basic built-in datatypes, such as int, char, etc, cannot be treated as classes in C++, and thus extended from. By using the constructor inheritance that DECO offers, it would then be possible to create new classes that extend or alter the operations available to basic integers, for instance. Currently, the only way to do this is to create an integer emulation class that has all of the same operations, which is not only redundant, but with most compilers generates less efficient code than just using the int type.

‡ This depends upon having access to the class and thus knowledge of the implementation. This would be equivalent to conventional C++ method calls.

interfaces will be created automatically. In such cases, for efficiency reasons it is generally better to use a const reference to the type, so that new type interfaces are not always created when the same types are being used. Looking back at *Example 5-1*, it can be seen that such a reference was declared with a typedef as t_str_. So, using that, the following function can be declared, which will take any string with any implementation, add it to itself, and return the new string length:

```
int
string_double(t_str_ s)
{
    s.append(s);
    return s.length();
}
```

If this is invoked from any of the following code, it will work as expected:

```
string_double(ds1);
string_double(D_String(s1));    // using Example String1.dc only
string_double(D_String("just a test."));
```

Before moving on to the next example, it should be pointed out that a proper interface for strings should be separated into at least three types, one for reading strings, one for writing them, and one for modifying them. Then, a general purpose fourth type can be created that combines the readable, writable, and modifiable string types. This separation is important for getting the most flexibility out of a system without sacrificing correctness.

5.10.2 *Sorting*

One of the main purposes of using *types* instead of *classes* is so that code can be written in a much more algorithmic sense, and not have to be rewritten when the data it acts on changes. However, this can be accomplished already in C or C++ by just using macros, if it is only the textual source code that needs to remain constant. Templates in C++ can also accomplish this in a more type-safe manner. The drawback to this these techniques is that in both cases the code must be recompiled for each time it is parameterized with a new *datatype*, and if multiple datatypes are used to parameterize the macro/template in a given program, multiple compiled versions of the code must exist, often causing excessive code bloat.

By using type interfaces to solve the problem, the code only needs to be compiled once. As an example, see *Example 5-4*, ¤ which implements a quick sort algorithm that sorts data represented by the t_sortable type interface. That interface is defined as follows:

```
type t_sortable {
    int cmp(int i, int j);
    void swap(int i, int j);
};
```

Only two pieces of functionality are needed to do a sort, (i) the ability to compare the keys at two

¤ This code is loosely based on the quick sort algorithms defined in [*Oliver93*] , adapted to DC++, and simplified to remove complexity for the sake of exposition. The original algorithm used an insertion sort when less than 10 items remained.

Example 5-4: `qsort.dc`

```
#include "t_sortable.dh"

void
qsort(t_sortable_ v, int lo, int hi)
{
    while(hi-lo >= 2) {
        int i;
        int m = (lo+hi)/2;
        if(v.cmp(m, hi) > 0)
            v.swap(m, hi);
        if(v.cmp(m, lo) > 0)
            v.swap(m, lo);
        if(v.cmp(hi, lo) < 0)
            v.swap(hi, lo);
        for(i=lo, m=hi;; ) {
            while(v.cmp(++i, lo) < 0) ;
            while(v.cmp(--m, lo) > 0) ;
            if(i >= m)
                break;
            v.swap(i, m);
        }
        v.swap(lo, m);
        if(m-lo < hi-m) {
            qsort(v, lo, m-1);
            lo = m+1;
        } else {
            qsort(v, m+1, hi);
            hi = m-1;
        }
    }
    if(hi-lo == 1 && v.cmp(lo, hi) > 0)
        v.swap(lo, hi);
}
```

specified locations in the array or list, and (ii) the ability to swap the data at two specified locations. Therefore, that is the only functionality put into this interface. Type interfaces need to be kept as simple as possible for their intended purpose, so that the implementation is not required to implement functionality that will not even be used.

Once this `qsort` function is compiled, it should never need to be compiled again. ◊ All that needs to be done in order to use it is to bind the implementation of some array or list of items with the `t_sortable` interface.

This brings up an interesting notion. For a given object, there may be more than one *key* for which sorting may be performed. Obviously the `t_sortable` interface is bound to an implementation that only uses one particular key as the "correct" one to sort by. This means that some technique is needed to bind an implementation to a type in more than one way. This technique applies directly to real world objects as well. It is possible to take an object and sometimes be able to use it for a given purpose in more than one way, although some ways are usually better

◊ Assuming that `int`s are adequate for storing the indices. If not, then a new type of sortable interface will need to be designed that uses a larger integer type, or even a numeric type interface. It was decided that the extra overhead of this was not necessary in most cases, since most current machines cannot store arrays with indices larger than their maximum `int` value.

than others.

There is currently no good solution to this problem in DC++, other than to create several hand-crafted dynamic bind functions and a means of selecting between them, so that the `t_sortable` interface can be bound to several different comparison functions in a single class.

5.10.3 *Circles and Ellipses*

Example 5-5 shows a way of defining types for circles, with the basic type for reading a circle, `r_circle` inheriting the methods from `r_ellipse`, the basic type for reading an ellipse. This makes `r_circle` a proper subtype of `r_ellipse`.

Example 5-5: `circle.dh`

```
#include "ellipse.dh"

type r_circle : r_ellipse {
    int radius() const;
};

type x_circle : r_circle {};

type w_circle {
    void radius(int);
};

type t_circle : r_circle, w_circle {};
```

Example 5-6: `ellipse.dh`

```
type w_circle;  // forward declaration

type r_ellipse {
    int major() const;
    int minor() const;
};

type x_ellipse : r_ellipse {};

type w_ellipse {
    void axes(int major, int minor);
w_circle:
    void radius(int r) { axes(r*2, r*2); }
};

type m_ellipse {
    void major(int);
    void minor(int);
};

type t_ellipse : r_ellipse, w_ellipse, m_ellipse {};
```

Looking at *Example 5-6*, it can be seen that `w_ellipse` implements {w_circle} which means that instances of implementations of `w_ellipse` can be seen as `w_circle` types.

Inheritance could have been used, but then the implementation of `w_ellipse` would be forced to implement the methods of `w_circle` as well. By making this an implementation instead of an inheritance, it still acts the same for subsumption, but allows a default implementation to be defined in the type.

Example 5-7: `Circle1.dc`

```
#include "circle.dh"
#include "ellipse.dh"

// Circle declared separately, just to show different
// techniques for building classes...
class Circle {
    int _radius;
public:
    Circle(int radius) : _radius(radius) {}
    int radius() const { return _radius; }
    void radius(int radius) const { _radius = radius; }
};


class D_Circle : public Circle {
public:
    D_Circle(int radius) : Circle(radius) {}
r_circle:
    int radius() const = D_Circle::radius;
w_circle:
    void radius(int) = D_Circle::radius;
r_ellipse:
    int major() const { return radius()*2; }
    int minor() const { return radius()*2; }
};
```

Example 5-7 shows a simple class `Circle`, which is then derived from to create `D_Circle` which implements the types `r_circle`, `w_circle`, and `r_ellipse`. This shows how pre-existing C++ classes can be used to just add a DC++ interface.

Example 5-8 shows a class `D_Ellipse` which contains its full implementation, rather than deriving it from an existing class. It implements types `r_ellipse`, `w_ellipse`, and `m_ellipse` directly. It also has a dynamic implementation for type `r_circle`. An attempt to bind to type `r_circle` with a conditional bind (?=) will cause the `dcheck()` function for that type to get invoked in order to see if the object is currently the correct type. If it succeeds, the implementation for `radius()` will be used to implement a readable circle using the ellipse implementation. This will only occur when the axes are identical, and the ellipse is indeed a circle.

Example 5-8: `Ellipse1.dc`

```
#include "ellipse.dh"


class D_Ellipse {
private:
    int _major;
    int _minor;
public:
    int major() const { return _major; }
    void major(int n) { _major = n; }
    int minor() const { return _minor; }
    void minor(int n) { _minor = n; }
r_ellipse:
    int major() const = D_Ellipse::major;
    int minor() const = D_Ellipse::minor;
w_ellipse:
    void axes(int maj, int min) { major(maj), minor(min); }
m_ellipse:
    void major(int) = D_Ellipse::major;
    void minor(int) = D_Ellipse::minor;
dynamic r_circle:
    bool dcheck() const { major() == minor(); }
    int radius(int) const { return major()/2; }
};
```

6

DECO

The DECO program takes C++ source files that have been enhanced with Dynamic C++ extensions (DC++) and converts them to standard C++ code. DECO can be used either as an IDL compiler, to convert just the header files that describe objects, or as a full compiler that converts DC++ into C++. [¥]

6.1 Input

The input to the DECO compiler is a DC++ program, as described in *Section 5 (The DC++ Programming Language)*.

6.2 Output

The output generated by DECO is standard C++ code, consisting of the original, non-dynamic C++ code, along with the automatically generated interfaces for the dynamic objects. The implementation of dynamic method calls and type conversions is included with `#include` directives or via directly inlined code.

6.2.1 *Diagnostics*

Errors, warnings, and other diagnostics are given in the following format:

filename`:`*linenum*`:` *error or warning*`;` *where*

[¥] As future work on DECO progresses, it will probably output C code, and then eventually assembly or direct machine code, as it becomes a more complete compiler.

An example error diagnostic is:

```
test.cc:12: no match for call `f(int)'; before `)'
test.cc:12: must correspond to one of:
test.cc:12:   int f(int, int)
test.cc:12:   int f(int, char *)
test.cc:15: 1 error; at end of file
```

An example warning diagnostic is:

```
old.cc:4: warning: implicit int; before `;'
old.cc:6: warning: function implicitly returns int; before `;'
old.cc:14: 2 warnings; at end of file
```

Errors prevent further stages of compilation from occurring, and will inhibit code output as well. Warnings do nothing other than print the diagnostic warning message, unless the `-Werror` option has been specified, in which case any warnings will cause the compilation to abort after parsing the current translation unit.

Diagnostic messages are issued for errors in the dynamic object specification, and for errors that can be caught simply by parsing. Many C++ errors are detected as well, although most errors that are caught are those that prevent building an internal tree structure of the program. Since the output is currently C++ code, many additional errors may be caught by the C++ compiler that the output is passed on to. Although this aspect of DECO is being gradually improved, it is not a primary focus of work on DECO. Therefore, it is sometimes difficult to locate the source of errors, since there is not a direct correspondence between the C++ output of DECO and the DC++ source input. When errors do occur in this second stage of compilation, it is probably best to try to locate and fix them in the output of DECO, and then go back and re-apply the fixes to the original source.

6.3 Options

The options to DECO are described in the DECO manual page, found in *Section B (DECO Manual Page)* on page 133.

6.4 IDL Usage

When DECO is used as an IDL compiler, DC++ header files (generally ending with a `.dh` extension) are taken as input, and result in two output files, a new header (a `.dh.h` file) and some C++ code (a `.dh.cc` file) containing run-time information about the types and classes in the original header file. The `.dh.h` file should be included from all code that needs to directly use the types and classes that were defined, including their implementations, and all such code needs to be linked with the compiled `.dh.cc` file. *Figure 6-1* shows this process.

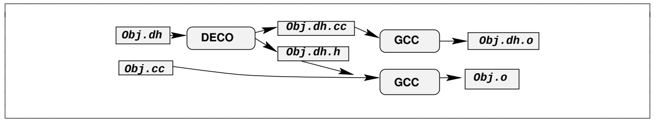

Figure 6-1: DECO IDL Usage

The commands needed to compile the example shown in *Figure 6-1* (in the absence of a Makefile) are:

```
% deco Obj.dh
% g++ -c Obj.dh.cc
% g++ -c Obj.cc
% g++ -o objtest objtest.o Obj.o Obj.dh.o -ldeco
```

When DECO is used as an IDL compiler, it is possible to specify lines that are passed through without pre-processing, so that control is available concerning whether lines normally handled by the C pre-processor, such as `#include` directives, are executed before DECO, or during normal inclusion of the resulting header file.

6.4.1 *Interfacing C++ with DC++*

There are several ways of interfacing C++ with DC++. One way is to just link together several objects files that were compiled separately, such that the DC++ code simply calls into the C++ code. This is, of course, the easiest way and is similar to interfacing any other language with "native methods" written in C or C++. However, it is sometimes desirable to have the C++ code call into dynamic objects written in DC++. In order to do this, the C++ program needs some knowledge of the implementation of DC++ objects, and also needs to be able to bind type interfaces to objects.

The following macros are provided for compatibility between C++ and DC++. These macro definitions will expand to either the standard DC++ code described in their definitions, or will expand to code that will be functionally equivalent in C++. This allows C++ programs to make use of dynamic classes and types, and also allows DC++ programs to include headers or snippets of code that use these macros without any problems. Without these macros, it would not be possible to use DECO as an IDL compiler, since the code that included the generated headers would have no way to utilize them.

`DBIND(`*t*`, `*r*`, `*o*`)`
: Equivalent to '`t r = o`'; for declaring a type interface and binding it to an object.

`DCBIND(`*t*`, `*r*`, `*o*`)`
: Equivalent to '`t r ?= o`'; for declaring a type interface and conditionally binding it to an object, leaving it as nil if that type is not supported.

`DREBIND(`*r*`, `*o*`)`
: Equivalent to '`&r = o`'; for re-binding a type interface after it has been previously declared.

`DCREBIND(`*r*`, `*o*`)`
: Equivalent to '`&r ?= o`'; for conditionally re-binding a type interface after it has been previously declared. This leaves *r* as nil if that type is not supported by the object.

`DASSIGN(`*r*`, `*o*`)`
: Equivalent to '`r = o`'; technically, this is not necessary, since it evaluates to this for both DC++ and C++. This was provided only for consistency, and probably does not need to be used.

`DCASSIGN(`*r*`, `*o*`)`
: Equivalent to '`r ?= o`'; this performs a conditional assignment of the value of the object to *r*. This only makes sense if *r* is some form of container (writable) type.

It is possible to define some of the methods of a DC++ object from a regular C++ file. This may be necessary when C++ features are being used that DC++ cannot currently handle, such as templates. In order to do this, it is necessary to know the translation of names from DC++ to regular C++. Implementations of types are named as typename__function. For instance, if you had the following DC++ class,

```
class Blah {
t_foo:
    int bar();
};
```

the implementation for `bar` in a normal C++ file would be written as:

```
int
Blah::t_foo__bar()
{
    return 42;
}
```

6.5 Compiler Usage

When using DECO as a full compiler, DC++ header files (`.dh` files) are just directly included from DC++ source code (`.dc` files). The entire code is passed through DECO and converted to standard C++, and the result can either be output, and later passed through a compiler separately, or it can be directly passed to the default C++ compiler (currently gcc) by DECO, making DECO act as a full compiler front-end. *Figure 6-2* shows DECO being used as a compiler.

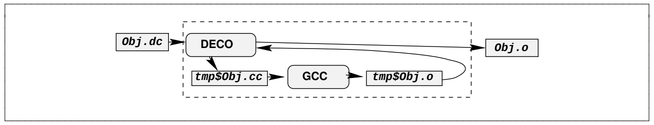

Figure 6-2: DECO Compiler Usage

To compile the example shown, the following commands could be executed:

```
% deco -c Obj.dc
% g++ -o objtest objtest.o Obj.o -ldeco
```

Or, to allow DECO to invoke the compiler directly, the following could be done:

```
% deco -o objtest objtest.o Obj.dc
```

Note that in the second example, the deco library did not need to be specified, since it was implicitly included.

6.6 Standard Files

In addition to standard header files and libraries, the following files are utilized by DECO:

deco The DECO compiler itself (binary executable).

dchdr.dh Internal code needed by DC++ to implement types, along with macro definitions for `type`, `dtype`, `dclass`, etc. This file is automatically included when using DECO to compile DC++ code.

dchdr.hh This file is the C++ equivalent of *dchdr.dh* and must be included before including any header files generated by using DECO as an IDL.

dcobj.hh Definitions of all data structures used by the DECO object framework. This file must be included in order to compile code that implements dynamic types and classes.

6.7 Internals

DECO is internally composed of a bunch of interactive components as shown in *Figure 6-3*.

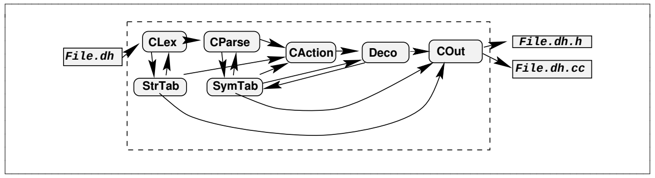

Figure 6-3: DECO Internals

`CLex`, `CParse`, `CAction`, and `COut` are all generalized classes for C++ parsing and output, and `StrTab` and `SymTab` are general-purpose classes for string and symbol tables. The `Deco` class is specifically designed for the implementation of DC++ used by the DECO compiler. All

of these classes are described in detail in *Section C (DECO Implementation)*, starting on page 139.

6.8 Limitations and Restrictions

A C++ compiler is required that conforms to most of the specifications of the C++ language in the ISO C++ Draft Standard. [*Koenig95a*] [*Koenig96*] [*Plauger95*] Under Unix, it was originally developed with GCC 2.7.2, and with a bit of work, it now appears to compile and run correctly with GCC 2.8.1, egcs-2.90.29, and egcs-2.91.66. It may not work with other C++ compilers until such compilers have adopted more of the new standard. Even then, there may be some complications with the auto-generated code that builds tables of method pointers.

6.9 Future

Throughout the development of the DC++ object model, and the implementation of DECO and ShagOS, many changes were made, and many times the entire design was discarded and replaced with a whole new model. This is part of the reason for separating the conceptual model from the implementation, and ensuring that DECO will handle the conversion from the conceptual model to an implementation in C++ code. This allows changes to be made to the implementation, or for optimized implementations to be written for specific architectures, without requiring the DC++ code to be rewritten.

Many changes have been made to the way the DC++ language is specified and implemented, and many more changes will probably continue to be made in the future, as the experimentation shows variations that are either necessary, more sensible or more efficient.

6.9.1 *Exceptions*

Exceptions and exception handlers are very useful for simplifying the process of writing robust code, and certainly they should be part of a complete dynamic object-oriented system. At this time, however, they have not yet been dealt with in conjunction with this work. Instead, error objects can currently be used; these will cause a program to abort if they go out of scope without being handled. They can be safely returned or copied, however, without causing an abort. [@]

To support a fully dynamic exception system, exception objects need to be dealt with in terms of their *types* rather than their *classes*. The C++ exception handling system, like the rest of C++, deals with exceptions based on their *class*. Therefore, `catch` statements in C++ cannot be used to provide this behavior, and code such as the following will not necessarily behave as expected:

[@] Essentially, they keep track of whether or not they were accessed or copied. Whenever they are accessed or copied, a flag gets set. If their destructor is executed and the flag is still clear, an abort is caused. Each new copy has the flag cleared again, so every copy must either be copied again, or be accessed. It is assumed that if information in the error object is accessed, the error is being correctly handled.

```
try {
    do_something_that_can_fail();
}
catch(t_warning w) {
    w.tell_someone();
    return true;
}
catch(t_commonError e) {
    e.deal_with_it();
    return false;
}
```

The reason this would fail is that the `catch` in C++ expects to catch an object of the specified class or any subclass of it, in the C++ sense of subclass. The type interfaces generated by DECO do not fall into the standard C++ class hierarchy, thus causing this to fail for subtypes.

The following code could be used to simulate a dynamic `catch`, and correctly implement the previous example:

```
try {
    do_something_that_can_fail();
}
catch(Obj o) {
    if(t_warning w = o) {       // caught a warning
        w.tell_someone();
        return true;
    }
    else if(t_commonError e = o) {  // caught a common error
        e.deal_with_it();
        return false;
    } else
        throw;      // we don't know what it is
                // so just pass it on...
}
```

This example assumes that the exception handling mechanism is fully-implemented enough to handle the run-time type information (RTTI) necessary to detect that a `catch` clause for class `Obj` should catch objects of any class derived from `Obj`. If this is not so, then `throw` expressions should be modified to statically cast the dynamic exception object to class `Obj` before throwing it. This is perfectly safe, since all `catch` clauses for it need to dynamically check its type and bind it before using it.

Of course, DECO could detect abstract types used as parameters for a `catch` clause when compiling DC++ programs, and could automatically generate code such as that above. Such an implementation would probably cause significant bloating of the final object code, however. A more efficient approach would involve incorporating an understanding of dynamic objects and types directly into the exception-handling mechanism.

6.9.2 *Templates and Genericity*

With the addition of dynamic classes and types to C++, completely generic functions and classes can be written, which simply perform according to their specifications, and make no restrictive assumptions about their parameters. This means that an array type and corresponding class can be created that can store objects of any arbitrary type in an array. While this is certainly very

flexible, it is sometimes too generic for some purposes.

As an example, if an array is to be created that can store employee records, which are of type `t_employeeRecord`, and it is guaranteed that nothing but an object of type `t_employeeRecord` is to be kept there, then it seems ridiculous to create just an array of dynamic objects. The generic array has at least two problems, which are closely related:

- Objects of other types can be put into the array
- When retrieving an object from the array, its type must always be dynamically checked, since the array by itself provides no type information.

Essentially, the problem here is that complete *genericity* based only on the union of all types loses too much compile-time type information, and thus the program must work harder at run-time to regain that information. It also means that a check for program correctness cannot occur at compile-time.

A partial solution to this problem is to explicitly create different types of arrays, for storing different types of objects, and explicitly create different functions, such as sorting algorithms, for sorting specific types of entities. Unfortunately, this involves writing repetitious code, and puts the burden of handling each situation back on the programmer, which is something that the dynamic object model was meant to eliminate, or at least reduce.

Another solution is the use of *parameterized types*, known in C++ as *templates*. C++ templates provide the ability to define a function or class once, and have it specialized with specific type information when needed.

Adding some form of template support to DC++ should probably be done at some time, but it is a very complex issue, and many ambiguities arise in both the syntax and semantics of standard C++ templates, to the extent that adding template support into DC++ and DECO could turn out to be a very challenging and time-consuming task. Also, templates may not be the best solution to the problem of adding genericity. #

Palsberg and Schwartzbach describe a system of genericity that fits in better with normal inheritance and subtyping schemes by using a modified version of inheritance with new invariants. [*Palsberg94*]

7

Implementation

The *Dynamic Encapsulation Model* was designed in such a way that the actual implementation used could be very flexible. In fact, implementations for different architectures can vary while still allowing object communication between the systems. However, without exposition of a concrete implementation, it is hard to show the validity of the model, and certainly impossible to demonstrate it in practice or make use of it. This section will show the standard implementation that was created for use on most 32-bit machine architectures.

There are two major portions to the implementation of the DEM. The first is the type interface, and the second is the standard representation used for objects and for interfacing with class implementations. Once the type interface is defined, then as long as it does not change, various new forms of the class implementations can be used, and even the default object format can be altered without requiring recompilation of any code that is based entirely on the type interface. Both the type and class implementations will be discussed in detail.

7.1 Requirements

This version of the DEM implementation will be limited to machines with at least the following characteristics:

- 32-bit standard word length
- 32-bit addressing range (pointers are implemented with 32-bit unsigned integers)
- ability to access bitfields (either intrinsically, or through some combination of shifting and masking)
- 4-byte alignment of pointers

The initial implementation was developed on several different Intel *x*86-based machines (where $x \geq 3$), □ and has also been ported to run on UltraSparc and MIPS-based machines.

□ Specifically, the Intel systems that this was developed and tested on include a 386SX-16, 486SX-33, 486DX2-66, and finally a Pentium II/350 during the last few months of the work.

For supporting 64-bit architectures in the future, the standard descriptors will need to be redesigned, but the general model should still work. It may be desirable to use 32-bit pointers for most objects even on a 64-bit machine, to conserve memory when large addressing spaces are not needed.

7.2 Type Interface

The type interface allows class implementations to be abstracted and utilized in a consistent way. All method calls must go through this interface, and it is from this interface that the Dynamic Encapsulation Model derives most of its flexibility. As such, the design of this interface is more important than any other aspect of the system. For future portability on any given processor architecture, this interface needs to be standardized. Its efficiency will also have a direct impact on all method calls made.

7.2.1 *Dynamic Type Binding*

Before any object can be accessed, or operated upon, a type interface must be bound to it, to make the implementation of the desired features of the object available. *Figure 7-1* shows the interface structure used for dynamic type binding.

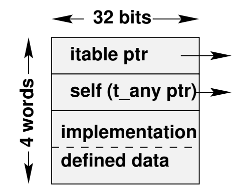

Figure 7-1: Dynamic Type Binding Interface Structure

This interface has two major components that are required:

- an *itable* pointer
  This points to an implementation table which supplies function pointers for all operations supported via this type interface.
- a *self* pointer
  This points back to a type interface for the original *self* that a call was invoked on. This is used when conversions have taken place but it is still necessary to get back to the original object for some operations.

This interface is known as a `TRef` or Typed Reference to an object. Every TRef also has an *unbound* state, signified by a nil value for the *itable* pointer.

The remaining two words (64 bits) are available to the implementer of the object to use for any purpose. This leaves much flexibility, since code using this type interface does not have any concern for what is stored in the second half of the `TRef`.

7.2.1.1 *Implementation Table*

The **itable** (implementation table) is an array of function pointers, with each one corresponding to a single method for implementing a type. For supertypes, an entry is reserved to point at the itable for each supertype. Binding a `TRef` to an implementation causes the `itable` field to either point to an existing itable, or to point to one that is then created and filled in appropriately. All method calls made to the object are then indirected through the itable to call the correct function. The functions all take the address of the `TRef` as the first argument.

All supertypes, no matter how many levels deep they are nested, have a direct pointer to their itable contained at the current level. This is a compromise between keeping each subtype distinct and merging all function pointers into a single flat table, in order to allow multiple inheritance with reasonable performance while keeping the table size reasonable as well. Having a pointer to the supertype itables, rather than including the method pointers directly, keeps the size of the itables from growing out of control as many levels of inheritance are used. Keeping a direct pointer to each supertype increases the size somewhat, but reduces the run-time cost and the size of the code performing the call by limiting all calls to at most three levels of indirection, those being the pointer into the itable, the pointer to the supertype, and the function pointer itself. This allows multiple inheritance while keeping the table size reasonable.

If a supertype has only one method and no inherited methods, then the function pointer for that method is directly substituted instead. This eliminates an extra level of indirection that would have saved a single entry only to replace it with a pointer to that entry. Since this would have had no practical value, the supertype's sole function pointer is inlined instead.

As an example, the following type definitions define some simple types for observing and changing a switch:

```
type r_switch {
    bool is_on();
};

type w_switch {
    void on();
    void off();
};

type t_switch : r_switch, w_switch {};
```

There is no `x_switch` defined, because we do not care about having a separate extensional concept of a switch in a particular state. A simple boolean variable suffices for that. Also, there is no `m_switch` because there is only one simple piece of state for the switch, so there is no practical reason to be able to modify the switch based on its current state. ⊕ The itable that gets generated for each type would effectively be the following, defined with C++ structures:

⊕ An exception to this might be if someone wanted a `flip` method that toggled the state without knowing what it currently was.


```
struct r_switch::itable {
    bool (*bind)(TRef*, TRef*, t_type_);
    bool (*is_on)(TRef*);
};

struct w_switch::itable {
    bool (*bind)(TRef*, TRef*, t_type_);
    void (*on)(TRef*);
    void (*off)(TRef*);
};

struct t_switch::itable {
    bool (*bind)(TRef*, TRef*, t_type_);
    bool (*is_on)(TRef*);
    w_switch::itable *w_switch;
};
```

Now if we want to add a dimmer to the switch, but still maximize substitutability of the types, we might define some new types as follows:

```
type r_dimmer : r_switch {
    int percent();      // returns 0 to 100
};

type w_dimmer : w_switch {
    void percent(int n);    // n = 0 to 100
};

type t_dimmer : r_dimmer, w_dimmer {};
```

Once again, no `m_dimmer` type is needed, since we do not need a way to change the dimmer based on its current value, although a real version of this might add methods for increasing or decreasing the setting instead of setting it to an absolute.

The itables for these new types would be:

```
struct r_dimmer::itable {
    bool (*bind)(TRef*, TRef*, t_type_);
    int (*percent)(TRef*);
    r_switch::itable *r_switch;
};

struct w_dimmer::itable {
    bool (*bind)(TRef*, TRef*, t_type_);
    void (*percent)(TRef*, int);
    w_switch::itable *w_switch;
};

struct t_dimmer::itable {
    bool (*bind)(TRef*, TRef*, t_type_);
    r_switch::itable *r_switch;
    w_switch::itable *w_switch;
    r_dimmer::itable *r_dimmer;
    w_dimmer::itable *w_dimmer;
};
```

Note that the last itable definition, for `t_dimmer` includes all the base classes directly, even those that were not defined at this level.

7.2.1.2 *Self Pointer*

Normally the `self` pointer points back to the current `TRef`, but when delegation occurs, the `TRef` that is passed on to the delegate will have a new `itable` which represents the object that control has been passed to, and a `self` pointer that points to the object that was originally called.

The reason for keeping the original `self` as a separate pointer is that it is often necessary for delegated actions to call other methods on the object, and these need to be invoked on the original object to be semantically correct. For instance, suppose a `Window` object is created that has both a `resize()` and `redraw()` function, and the `resize()` function calls `redraw` when it is finished. If a `TextWindow` object is then created that uses `Window` as its parent or `superobject`, and if `TextWindow` provides a new implementation of `redraw`, then a potential problem exists. Obviously, calling `resize` ought to redraw the `TextWindow` object, but delegation passes the `resize` call from `TextWindow` on to its parent, `Window`. If `resize` simply invokes `redraw` on the *here* object (which would be `Window`), then the `TextWindow` does not get redrawn. What needs to happen is that the `redraw` method needs to get invoked on the `TextWindow` object, which is the original object the current call was invoked on, also known as the *self* object. This is why most implementations should re-bind *self* and utilize it rather than using *here* whenever possible. *

7.2.1.3 *Invoking Methods*

The code for calling methods can either be generated inline for each call, or could be deferred to a general stub for each method of each interface type. In the direct inline case, the following actions need to be taken for each call:

- push the address of the `TRef` as argument 0 (or put it in a register, depending upon the standard calling conventions for this architecture)
- push the rest of the arguments needed for the method call
- get the itable pointer from the `TRef` structure (it is the first field in order to make this lookup as fast as possible)
- if the method is part of a supertype, and that supertype has more than one direct method or has another supertype (i.e. is not inlined), then get a new itable pointer from the appropriate supertype pointer in the itable
- use the itable pointer and the method index number to find the appropriate function pointer
- call the function

* `this` in C++ is a hybrid of the `self` and `here` concepts. It only truly acts as `self` for virtual member functions. For static member functions and for all data accesses, it acts as `here`.

7.2.1.4 *Type Conversions*

When a bound object needs to be converted to a supertype of the type it is currently bound to, a new `TRef` can be created without even calling `bind` again. All that needs to be done is to take the new `TRef` structure and fill it in with almost the same information, adjusting the itable pointer to the address of one of the supertype itables. For supertypes that had only a single method, the supertype itable needs to either be located or constructed. For this reason, and for resolving some other more complex issues pertaining to the memory management for itables, it does become necessary to call `bind`. However, the `bind` function can be tailored to the type so that it reuses constructed itables as much as possible, rather than recreating them.

If the code makes its own local copy of a TRef, it is responsible for maintaining it, keeping track of reference counts, etc, and ensuring that the original TRef passed to it is still valid, thus an extra bind may be required to "hang on" to an object reference.

Conversions to types that are not direct supertypes are more complicated, sometimes involving extensive searches of the type hierarchy, and dynamic generation of itables, to provide compatibility for types that are able to be used interchangeably, but were not originally designed to be so.

7.2.2 *Itables vs Vtables*

At first, it seems like it would have been easier to use C++ *vtables* instead of inventing a new variation of function pointer table for implementing the type interfaces. Vtables are the standard technique used by many C++ compilers to implement virtual function calls. In fact, using vtables was tried repeatedly as a way of implementing abstract types, with the thought that it would be more straightforward to make use of similar functionality that already existed, but there always seemed to be complications.

In the end, *itables* were used instead of vtables for the following reasons:

- Multiple inheritance causes multiple vtable pointers to get added directly to the object's instance data. In this case, that would be the `TRef`, which is supposed to have a predefined fixed size. This would be inconsistent with the DEM.
- Vtables were created for C++, and the binary type interface on a given architecture is supposed to be language independent.
- There is no guarantee that multiple C++ compilers on the same architecture will have the same vtable format. In fact, different versions of the same compiler could change the vtable format. This would require a recompilation of all code to be consistent, and would prohibit communication between code generated by different compilers.
- The standard vtable layout has extra information that is not necessary for the DEM, such as run-time type information that can be gathered via type-binding tests.
- Some means of ''hacking'' around the standard vtable implementation would have been necessary anyway, since the tables sometimes need to be dynamically built. This workaround could cause incompatibilities with different compiler implementations.

7.2.3 *Previous Variations*

Previous implementations of the DEM (prior to it being named such) used a different manner of invoking functions from the itable. There was originally a different layout for the `TRef` that looked like the one in *Figure 7-2*. The `TRef` was called a `BTRef` at that time. The 'B', which stood for 'Bound' has since been dropped since `TRef`s can clearly have an unbound state as well.

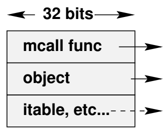

Figure 7-2: Old BTRef Structure

In this example, the object pointer contained the address of an Object Descriptor, which had a known format for describing the object the interface was bound to. The Object Descriptor was similar to the one that will be described in *Section 7.3.1 (Objects)*, but since it was directly pointed to here, it was required to be of a known format, thus making a fixed binary implementation of the object descriptor also a requirement for compatibility. The new scheme avoids this requirement.

The `mcall` function pointer was a pointer to a generic function for performing method calls for this type. This function was called with the first argument being a pointer to the `BTRef` structure, as it is now, but the second argument had to be the method number. This meant that the function that was called here had the responsibility of then taking all of the remaining arguments, whose number was potentially variable, and arranging them properly for invoking the specified method. This was tricky, non-portable, and not necessarily efficient. On some architectures, this could be very difficult to do in a general way that allowed it to work for any method call.

The third field of the old `BTRef` structure was generally used for pointing to the address of the *itable*. Two standard forms of *itables* used to exist. The first form, as shown in *Figure 7-3*, simply had function pointers. Another variation, known as the **p-itable**, or *parameterized* itable, had 32-bit data values along with each function, as shown in *Figure 7-4*. These values were always passed as the first argument to each function. The advantage of supplying such data in the table was that one function could be used to implement multiple methods, or even methods for different types of objects, by simply providing an argument to distinguish the implementations, when it was practical to do so. This functionality can also be implemented with non-parameterized itables, and thus also with the current form of itables, by using *thunks*. †

One of the big advantages to the original `BTRef` layout was that the implementation did not have to use itables if another technique was more appropriate. In addition, the code generated for the method calls was kept very simple, since it just pushed the address of the `BTRef` as the

† **Thunks** are small blocks of interface support code that were originally used in Algol60 to implement call-by-name semantics. In this case, they would be used to get the argument value, and would then call the parameterized function. [*Pratt84*]

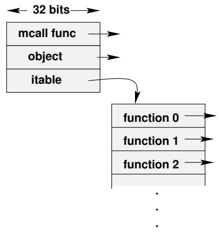

Figure 7-3: Old-style Interface Table (itable)

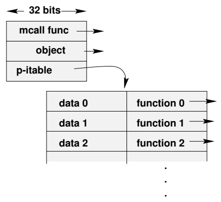

Figure 7-4: Old-style Parameterized Interface Table (p-itable)

first argument, followed by the method index, and then the rest of the arguments. It then made an indirect function call through the pointer at offset 0 in the `BTRef` structure. The new way of making method calls is much more complex and less flexible in comparison, although the code for the call can be kept small by not inlining the stubs used for the interface. The two main reasons for switching were the difficulty in making the `mcall` implementations even remotely portable, and the potentially high run-time cost associated with most `mcall` implementations. The new technique has a much more direct call chain.

Of course, now additional code is required if a generalized notion of a method call is needed; for instance, when method calls need to be passed to remote machines or address spaces. In such cases, a set of specialized stubs are used as the implementation. These auto-generated stubs are available for every interface type, and they simply push the following onto the stack:

- the method index
- a pointer to a `TRef` for the method descriptor

- a `va_list` ‡ for the arguments

They then invoke a function specified by one of the implementation-defined `TRef` fields. Advanced linker support will be necessary to strip all of the unused stubs out of the final executable code; otherwise, an excessively large executable could result.

7.3 Class and Object Implementation

Since the type interface provides the standard through which all method calls are invoked, the actual implementation used for classes and objects can be changed without affecting compiled code that uses the type interface. In fact, multiple class and object implementations can coexist in the same system. Nevertheless, it is useful to have a standard way of implementing objects and classes to reduce the work required to develop new objects. Also, with DECO, the conversion from DC++ code to actual object implementations is supposed to be automated. This automation cannot occur without a standardized representation.

7.3.1 *Objects*

Objects are the foundation block on which everything else is implemented. Types and dynamic classes have objects that describe them. Therefore, the generalized object will be presented before describing particular types of objects.

7.3.1.1 *Object Descriptor*

All generic dynamic objects need an **object descriptor**, which is a simple fixed-size structure that describes the implementation class of the object, and either contains the data for the object, or points to where it may be obtained.

For a 32-bit machine, the object descriptor can be implemented as shown in *Figure 7-5*. Note that the low two bits of the class pointer (called *ptype*) are used to tell how to interpret the data pointer. This requires that all class descriptors are aligned on a 4 byte (32 bit) boundary, but this alignment is standard on most 32-bit architectures.

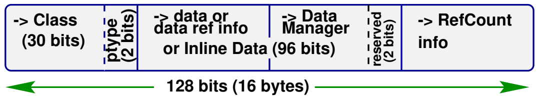

Figure 7-5: Object Descriptor

The *ptype* field can have the following values:

‡ `va_list` is the predefined datatype used to represent a variable list of arguments in C and C++.

0: Inline
The size of the data is less than or equal to 96 bits, and is stored in the remaining portion of the object descriptor.

1: Local
The data pointer has the 32-bit address of the data, and the data manager pointer contains the address of the object descriptor for managing the associated storage. The descriptor for the data manager, like the class descriptor, must be aligned on a 32-bit boundary. The two low bits of the data manager pointer are reserved for use by the data manager.

2: Remote
The data pointer is not a valid address, but rather some sort of pointer or other value that the data manager can use to locate the data if it is needed. The data can be thought of as ‘‘paged out’’ to some non-local storage if it is marked as *remote*. Alternatively, it could be local, but compressed, encrypted, or in some other manner currently unusable. The two low bits of the data manager pointer are reserved for use by the data manager.

3: Reserved
This is reserved for future extensions.

For a 64-bit architecture, the same general format can be used with 64-bit pointers and a 256-bit object descriptor, allowing up to 192 bits of inline data. On such machines, if 64-bit alignment also occurs, an extra flag bit may be available above *ptype*, for implementation-defined purposes. There will also be an extra reserved bit in the Data Manager pointer.

A 16-bit machine with 16-bit pointers could use this general format with a 64-bit descriptor, and have up to 48 bits of inline data. However, on a 16-bit architecture, 32-bit alignment of the class descriptor would need to be forcibly ensured, to make both of the *ptype* bits available, and to be able to maintain the two reserved bits. Since object descriptors would be 64 bits in size, it should not be unreasonable to align them to 32 bit boundaries.

7.3.1.1.1 *Inline Data*

Object Descriptors were designed to handle inline data as the most optimal case, by making the *ptype* field equal to 0, as shown in *Figure 7-6*, so that no masking of the class pointer is needed. The remaining 96 bits are available to store the inline data. Note that simply because data is small enough to be inlined does not mean that it must be or ought to be.

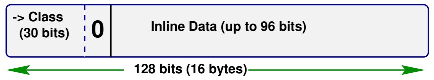

Figure 7-6: Object Descriptor for Inline Data

7.3.1.1.2 *Local Data*

All normal data that can be accessed via a standard pointer can be considered local data, which is referenced via the Object Descriptor shown in *Figure 7-7*. A *Data Manager* is associated with the local data, and is used to handle freeing of the data, or paging it out to some external device. If there is no data manager, a condition which is specified by setting the data manager pointer to 0, then the data is assumed to be automatic data, allocated on the stack.

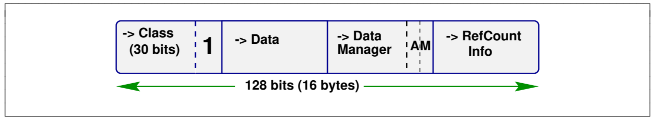

Figure 7-7: Object Descriptor for Local Data

The two low bits of the Data Manager pointer are used to flag accesses (`A`) and modifications (`M`) to the data. [¤] Every time the information pointed to by the data pointer is read, the "`A`" bit is set, and every time some operation causes a modification of the object represented by the data, the "`M`" bit is set. These can be used by the Data Manager to keep track of what data blocks were most recently used, and also to determine if the data needs to be written to a backing store if it is swapped out. It is important to note that the "`M`" bit is used to flag modifications with respect to the object that is represented, not just the raw data. Therefore, if reading an attribute causes the data to be modified in a way that does not alter the represented object, such as changing some cache entries to speed up future accesses, then this bit will *not* be set, and if the object is paged out, it will not be saved. This is due to the fact that, in most cases, the overhead of writing back the object is greater than recomputing such cache information, and since the new representation still refers to the same object, there is no need to save it. It is up to the implementation code to decide when to set these bits.

7.3.1.1.3 *Remote Data*

Remote data, which is referenced by the Object Descriptor shown in *Figure 7-8*, can be used to represent a variety of non-locally addressable storage, including:

- Data that has been paged-out to a disk
- Data that resides in the address space of another local task
- Kernel-protected data
- Remote data on another machine
- Compressed data

[¤] This means that the descriptor for the Data Manager, like all other object descriptors, must be aligned on at least a four byte boundary.

- Encrypted data

For all of these cases, the Data Manager serves a much more active role than it does for local data. Every access to the object must go through the Data Manager, in order to map a readable or writable form of the object. Note that the Data Manager has the option of either temporarily mapping such storage, or it may modify the object descriptor and convert the *ptype* to local data, thereby effectively swapping in the data.

Data Managers will be discussed in more detail in *Section 7.3.2 (Data Managers)*.

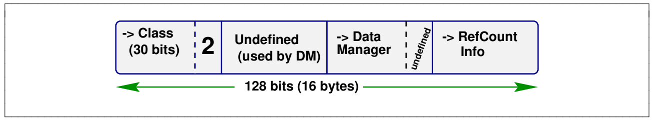

Figure 7-8: Object Descriptor for Remote Data

7.3.1.1.4 *Reserved*

The last data category is currently reserved for future use, since there are four possible values for `ptype` and only three (Inline, Local, and Remote) are currently needed. Perhaps this can be used in the future to distinguish between different forms of local or remote data, or for an entirely new, as yet unconceived, form of data storage.

7.3.2 *Data Managers*

Data Managers are objects whose purpose, minimally, is to be able to allocate blocks of data, and later free them. In addition, some Data Managers may handle requests for mapping in data and unmapping it, if the storage they represent is not directly-accessible local memory. This includes data that may be in local memory in a form that cannot be directly used, such as compressed or encrypted data.

Data Managers must support type `t_datamgr`, which has the following features:

```
type t_datamgr {
    bool valid(ObjDesc *) const;
    void alloc(ObjDesc *);
    void free(ObjDesc *);
};
```

The `valid` function can be called to determine if the data pointed to by the object is a valid pointer into the storage handled by the Data Manager. This is not guaranteed to be able to detect and return `false` for all invalid pointers, but it must be guaranteed to always return `true` for valid pointers, and should try to return `false` for as many invalid pointers as possible. Therefore it can be used to help eliminate at least some pointer errors before they become more serious. Calling `valid` with a null pointer can be used to tell whether this feature is supported at all. If it is not supported, it will return `true` for all pointers, including the `nil`.

In addition, Data Managers handling remote data must support type `t_rdatamgr`:

```
type t_rdatamgr : t_datamgr {
    void* map_r(ObjDesc *);
    void* map_w(ObjDesc *);
    void* map_rw(ObjDesc *);
    void unmap(ObjDesc *, void *);
};
```

7.3.3 *Object Managers*

Object Managers are used for accessing entire remote objects, rather than just remote data. Using a Data Manager, the class still has to reside locally, and the data has to be mapped in to be usable. With an Object Manager, there are no such rules. The Object Manager allows the object, and all of its implementation, to reside anywhere, allowing for fully remote objects.

An Object Manager is a special object which acts as the *class* for the objects that it manages. The data stored locally for each object is some sort of reference that the Object Manager knows how to use to locate the actual object.

Object Managers can be used for a variety of purposes, including:

- Objects in other address spaces (Inter-Process Communication)
- Compressed or encrypted objects
- Distributed objects (Remote Procedure Calls)
- Invoking methods of protected objects (i.e. System Calls)

Since most object managers have no intrinsic knowledge of the types of objects they are managing, they must be prepared to pass on all varieties of function calls and often have to convert the arguments to a standard format for shipping them off to other address spaces. Remote procedure calls, combined with flattening and packing, are used to solve this problem.

7.3.4 *Remote Calls*

Communication between address spaces or across networks is handled by creating proxy objects that perform the communication, and contain local state information on how to reach the other end of the connection. The currently bound type can be queried about the parameters for the method that was called, and each argument can then be *flattened* or *packed* to send it, depending on whether the call is to another object on the same or a different machine, respectively. For some arguments, especially those that are either *intensional* or very large, a proxy object may be created rather than passing a representation of the object.

There is no reason a unique itable needs to be used for an implementation. In fact, for calls to remote objects, such a table would be useless. Depending upon the communication protocol used, it may make sense to keep some sort of translation table, but in most cases the method number is simply passed to the remote object manager, which actually invokes the method call. What is needed is an itable that is specialized for each type only, not per implementation, consisting of stubs that can pass on the arguments in a generalized way along with the method index.

7.3.5 *Flattening and Packing*

Flattening is a way of taking objects of arbitrary complexity, whose representations may be scattered all over memory, and converting them into a simple flat representation that can be sent to other address spaces. This means all pointers must be followed and stored in the flat representation in a way that retains the relative structure of the data. For a linked list, the representation could simply be a single aggregate of all nodes in consecutive order. For a tree, the nodes may still need to be stored individually, perhaps using relative indices into the flattened image instead of pointers. Flattening is performed by calling the `flatten` method of an object, which is part of `t_flat` and has the following signature:

```
int flat(byte *buf, int len);
```

If the buffer is not big enough for the entire flattened representation, then the required size is simply returned. Thus, `flat` can be invoked with a length of 0 to test how large a buffer is needed. Needless to say, the object must not be altered between this test and the next call to `flat` or the size could change. If this could be a problem, then every call to `flat` must have the resultant size checked, possibly causing in another iteration if the result is larger than the length passed in.

Packing is a combination of *flattening* and *translation*. Since different machines store data using different representations, translation of the tokens is necessary in order for each token to still refer to the same object when it is transferred to another machine. Some trivial examples are changing the byte ordering of integer numbers (big-endian vs little endian), or the character representation for strings (ASCII vs EBCDIC). A more complex example is translating the identifier used for a given class, since this could potentially involve much additional communication to find the equivalent class on the other machine. This may, in fact, involve sending a copy of the class over to the other machine or generating a proxy to invoke it without duplicating it.

Packing is performed by calling the `pack` method of an object, which is part of `t_pack` and has the following signature:

```
int pack(byte *buf, int len);
```

This follows the same semantics as `flat`, except that translation occurs as well as flattening when creating the byte stream.

By default `flat` will invoke `pack`, and thus objects need only support `t_pack` and not `t_flat`. However, by supporting both separately, extra efficiency can be gained for local IPC, where translation is not necessary.

7.3.6 *Unpacking and Unflattening*

Unflattening and unpacking both involve calling the correct constructor for the specified object. Constructors for specific implementation classes can support the `unflat` method, which is defined as:

```
type unflat(byte *buf, int len, int *plen);
```

The return value is a bound reference to an object of the expected type for that constructor. If `plen` is non-nil, the integer pointed to by it is assigned the total number of bytes that was unflattened from the byte stream, allowing multiple objects to be flattened into a single image and correctly unflattened.

For unpacking, constructors for types must have the `unpack` method, which is defined as:

```
type unpack(byte *buf, int len, int *plen);
```

This follows the same conventions as `unflat`.

7.4 One Instance Rule

Object managers are responsible for ensuring that only one instance of each type or class object exists in any given address space. By doing this, it is possible to check classes and type descriptors for equality by simply comparing the pointers to the object descriptors, rather than checking for deep equality. This reduces some code size overhead, and greatly reduces the run-time overhead that would otherwise be spent trying to compare the UUID's of the objects.

A similar strategy is utilized to implement I-Strings, [*Wild97*] where it is used for comparing strings by insuring that only one actual instance of each string exists, so that only a pointer comparison is needed to compare two strings for equality. This ensures that deep equality implies shallow equality, instead of just the converse; therefore the contrapositive is also true, which is that shallow inequality implies deep inequality. Thus testing for shallow equality is equivalent to testing for deep equality, and this can be used to greatly boost efficiency by avoiding the extensive number of comparisons often needed for deep equality.

Of course, this means that object managers are required to search through all objects for a match whenever a new one is brought into their address space. This is considered reasonable, however, since they already have to keep track of all such objects, and objects (especially those that represent types) are generally compared far more frequently then they are moved between address spaces. There could potentially be a problem with this technique if multiple object managers in the same address space have different handles for the same object. Since no obvious example yet exists of how this would occur in a practical system, this problem will have to be tackled later if it really exists.

7.5 Multiple-Architecture Support

Support for multiple architectures could be provided by either a machine-independent byte-code or by supporting multiple architectures with native code. Although not yet implemented, this model was designed with such considerations in mind. The code that implements the `bind` for an object is simply responsible for filling in a function table, and therefore there is no reason it cannot fill it in with calls into a virtual machine, along with pointers to the individual blocks of byte code. Once executing inside the virtual machine, the transition is even easier, since the itable can actually consist of direct calls into other byte code, without requiring any stubs for crossing from machine code to interpreted code.

7.6 Performance and Efficiency

Bind/call performance with inlining bug

iterations per bind	simple call		field access	
	DECO/DC++	GCC/C++	DECO/DC++	GCC/C++
1	202207 μsec	91753 μsec	215405 μsec	110356 μsec
2	127605	59997	149950	77833
3	108218	49920	124103	65227
5	95157	51515	117210	66972
7	84450	46868	104675	59989
9	78513	42154	97745	57568
15	70638	38412	88410	52436
24	65501	36091	82836	49175
48	62966	33279	78514	46659
94	59474	32953	76417	45797
148	58698	32022	75618	45364
232	58444	31529	75106	45124
363	58352	32745	74085	45841
568	57959	32129	73621	45794
888	57599	32838	73749	45620
1388	57792	31974	74221	44778
2169	57731	32555	74250	44744
3390	57545	31573	73977	44446
5298	56573	32579	73935	44576
8279	57021	32858	74182	44735
12937	57397	32439	73069	45854
20215	56872	31638	73509	44464
31587	55647	31792	72681	44002
49355	56989	30917	73230	44257
77118	52805	30084	68288	41609
96398	55709	31087	70733	43196

Table 7-1: Binding/Calling Performance (with inlining bug)

The performance of both the DC++ type interfaces and the implementation stubs for DECO was tested by using the program *`perf.dc`* which is shown in *Section E.1 (Source Code)* on page 161. This program consists of the following four tests:

- simple DC++ call through a bound type-interface
- simple C++ virtual function call
- DC++ call with one argument and object data access
- C++ virtual function call with one argument and object data access

All of the tests were run for a total of 1,000,000 iterations, with the number of calls per bind being varied.

In order to run these tests, the *perf.dc* program was converted with DECO to C++, and run, as follows:

```
deco perf.dc > perf.cc
gcc -O2 -o perf perf.cc
./perf > perf1.dat
```

This performance test was run on a 360MHz [◊] Pentium II PC, running Linux 2.2.12, and using GCC version egcs-2.91.66 (egcs-1.1.2 release). An abridged table of the results are shown in *Table 7-1*. The full, raw, output can be found in *Section E.3 (Results)* on page 190.

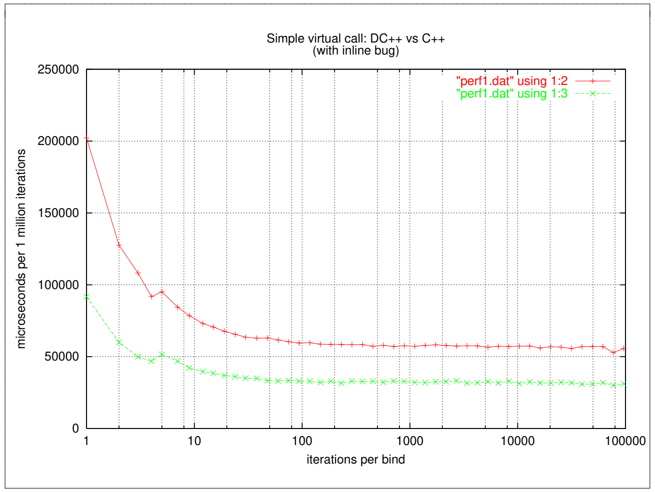

Figure 7-9: Binding/Calling Performance: DC++ vs C++, case 1

Figure 7-9 shows a graph of this data that compares the performance of a simple DC++ method call with a C++ virtual function call. As expected, the overhead is fairly high for the binding that DC++ requires, causing the calls to take over twice as long as standard C++, but as the number of calls per bind is increased, the average time drops to about a constant 75% higher than the C++ virtual function call. This was expected to be closer to 0%.

[◊] Yes, it is marginally overclocked. It is supposed to be a 350MHz processor, but the bus is running at about 103MHz instead of just 100.

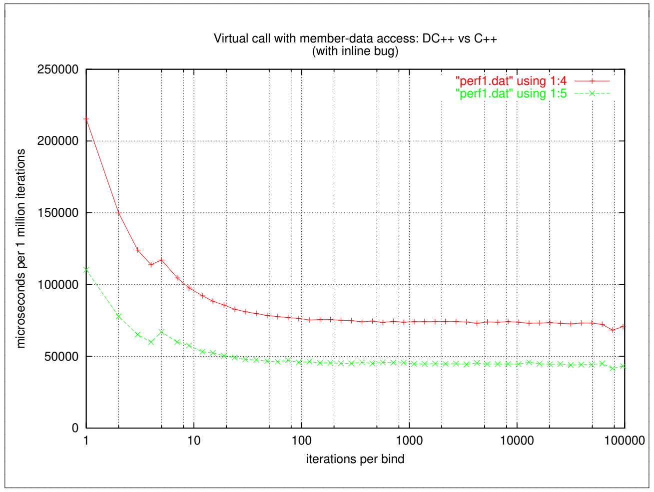

Figure 7-10: Binding/Calling Performance: DC++ vs C++, case 2

Figure 7-10 compares the performance of DC++ to C++ for a call which passes an argument and makes use of one field of the object's data. With this test, DC++ fares only about 65% worse than C++, but this was still expected to be 0%. *Figure 7-11* compares the performance of the simple and slightly more complicated DC++ calls, showing that there is difference in overhead for the bind in this case. *Figure 7-12* does the same for standard C++ virtual function calls, with similar results. All of these graphs clearly show that DC++ is paying a heavy penalty per call that C++ virtual functions do not have. A lot of work went into trying to create a mechanism that was equal to, or better, than a C++ virtual function call, so this came as quite a surprise.

After some further investigation it turned out that the lack of performance was the result of a bug in the output code of DECO. [¥]

[¥] This is the portion that is called `COut`; its implementation is described in *Section C.8 (COut)*.

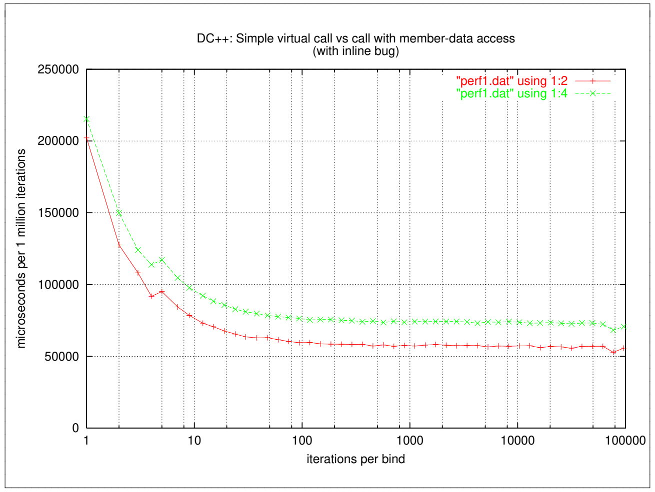

Figure 7-11: Binding/Calling Performance: DC++

The following commands were executed to see what was really happening: [@]

```
gcc -O2 -S -fno-exceptions -fno-rtti perf.cc
more perf.s
```

This assembly listing, which appears in *Section E.2 (Assembly Listings)* on page 176, showed that the inline code for the `f()` and `g()` functions of class `dc1` [#] were not getting inlined into the stubs that get put into the itable for the `dc1` implementation. Therefore, the DC++ code was incurring an extra "relay" call from the stub to the real function. As can be seen from the data, such a relay can be quite expensive for a very simple call.

At first it seemed very odd that the `f()` and `g()` functions were not being inlined, since they are declared as `inline` in the *perf.cc* file. Apparently the problem was that C++ (or at least the GNU g++ compiler) will only inline functions that are defined, not merely declared, before the code that calls them since the compiler acts fully on each function after parsing it, and cannot defer the inlining until later. It also appears that GCC will silently ignore any inline

[@] The `fno-exceptions` and `fno-rtti` flags have been added to keep the assembly listing to a much more reasonable size. Without them, the listing size doubles because tables of information that are not needed for this program are added. With past version of GCC, which were used during the development of DECO, exception handling and RTTI were turned off by default. With the newer ecgs versions, the defaults have been reversed.

[#] See the *perf.dc* source code for more information.

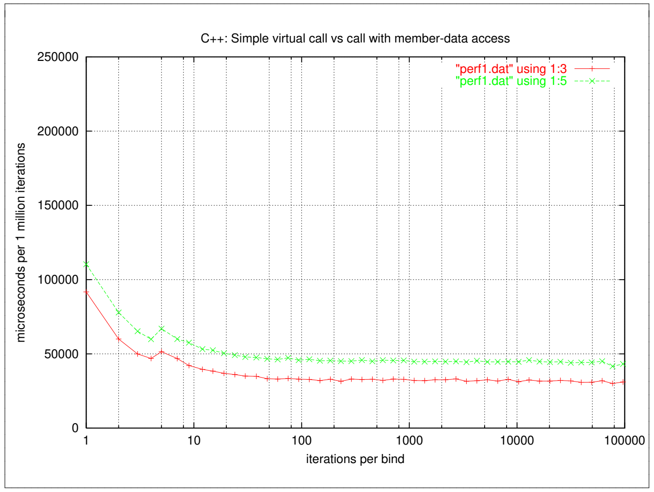

Figure 7-12: Binding/Calling Performance: C++

functions that are declared in the wrong order and instead just generates code that calls the function with a traditional function call. By looking at the output of DECO, *perf.cc* in *Section E.1 (Source Code)*, it can be seen that the definitions appeared in the wrong order.

An attempt was made to modify DECO (or more specifically, `COut`), to note this inline dependency when ordering the output, but it turned out that solving the matter correctly caused other complex issues to appear that needed resolving. Due to insufficient time to keep improving DECO, this fix has been deferred. Instead, as a temporary fix for the sake of performance testing, the DECO output in *perf.cc* was manually reordered, as shown in *perf2.cc* in *Section E.1 (Source Code)*. This was then recompiled, and run, with the following commands:

```
gcc -O2 -o perf2 perf2.cc
./perf2 > perf2.dat
```

This resulted in the data shown in *Table 7-2*.

Bind/call performance with bug fixed				
iterations per bind	simple call		field access	
	DECO/DC++	GCC/C++	DECO/DC++	GCC/C++
1	178615 μsec	91774 μsec	221749 μsec	110517 μsec
2	101291	59713	116808	78640
3	88371	50718	111772	67232
5	66944	51409	92016	68207
7	59147	46025	82613	60191
9	52920	42146	74718	57254
15	44183	38295	66202	52610
24	39606	35805	60002	49220
48	35990	33871	55748	46705
94	34004	32246	54079	45758
148	32654	32575	52372	46043
232	33070	31790	52557	45152
363	32475	32189	52333	45261
568	32669	32133	51509	45057
888	31660	32843	51405	45656
1388	31253	32842	51244	45672
2169	31217	33909	51973	44816
3390	31122	33133	52239	44452
5298	31049	32541	51758	44588
8279	30959	32881	51958	44639
12937	31050	32410	51720	45018
20215	30855	32172	51441	44490
31587	30526	31853	50087	44792
49355	31624	31728	50343	44649
77118	29769	29371	47931	41554
96398	30273	31025	49150	43368

Table 7-2: Binding/Calling Performance (with fixed inlining)

An assembly listing of `perf.cc`, called `perf2.s`, was also generated with the following command:

```
gcc -O2 -S -fno-exceptions -fno-rtti perf2.cc
```

It can be found in *Section E.2 (Assembly Listings)*. Examination of this assembly shows that the inlining problem is now fixed.

The data in `perf2.dat` was used to generate a new set of graphs with the inlining bug fixed. *Figure 7-13* shows that, although there is still a significant overhead for the binding operation (nearly 150 nanoseconds per bind, as compared to just 30 nanoseconds per method call), this overhead quickly disappears if the typed reference being bound is used repeatedly. In fact, after about 50 calls per bind, the total overhead is less than 5% as compared to a C++ virtual function. After 700 calls per bind, the overhead is gone altogether, and the DC++ method calls actually show themselves to be marginally faster than C++ virtual function calls.

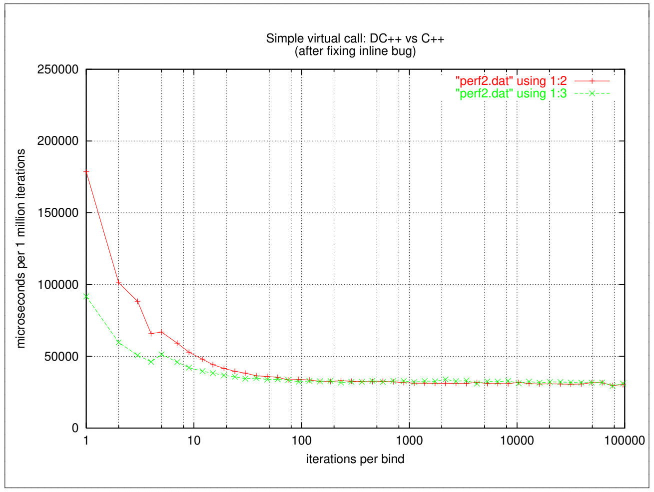

Figure 7-13: Binding/Calling Performance: DC++ vs C++, case 1 (bug fixed)

Figure 7-14 shows that DC++ does not fare quite as well for the more complex (but still rather simple case) of passing in a single argument and accessing a field from the object's data. *Figure 7-15* shows that these test cases take about 60% longer than the very simple ones. Of course, it should be noted that a multiplication has also been added, so the comparison in *Figure 7-14* is much more meaningful.

By looking at the assembly, it can be seen that the extra overhead shown in comes from accessing the field, since it has to go through a double indirection in order to obtain a pointer to the actual data for the object. In general, the overhead will still be negligible for most methods which actually perform more work, since even in this very simple case, DC++ method calls only take about 15% longer than C++ virtual function calls.

It should be noted that the "broken" case of DECO with the less efficient non-inlined code could still happen in real-world programs, since a lot of more complex methods will not be able to be inlined, for a variety of reasons. When this happens, the overhead will increase with the number of parameters since they all have to be copied onto the stack a second time to make the "relay" call. This could be better optimized with some custom inline assembly for the call, rather than just generating one C++ function that calls another. Also, a smarter compiler might be able to optimize the call to avoid copying the parameters a second time if they are just being passed on. This would be much easier on systems whose calling conventions allowed at least the first argument to be in a register so that it can be changed independently of the rest of the

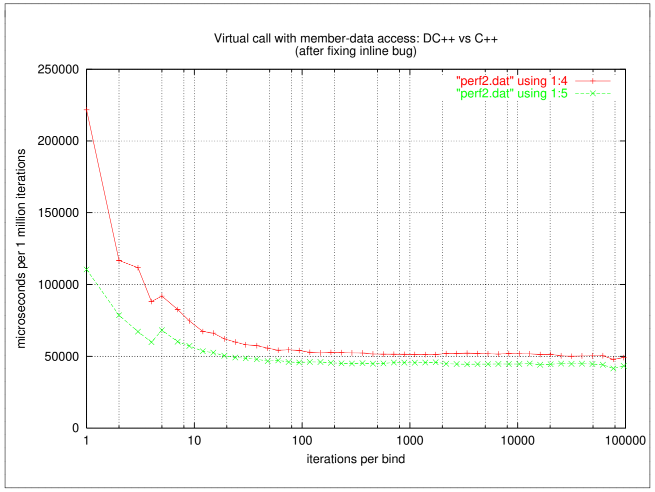

Figure 7-14: Binding/Calling Performance: DC++ vs C++, case 2 (bug fixed)

arguments on the stack.

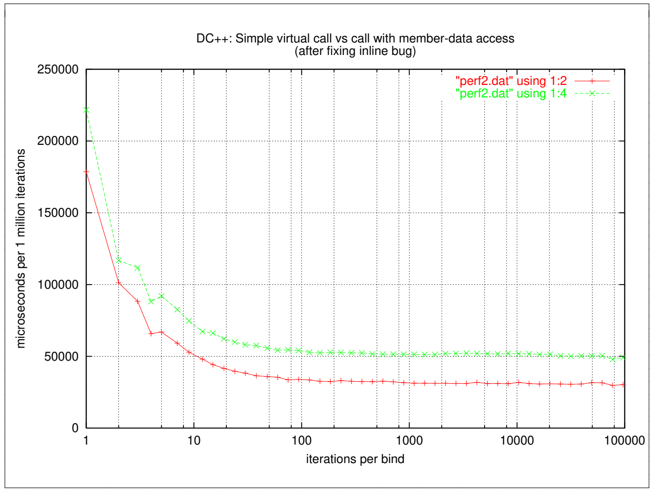

Figure 7-15: Binding/Calling Performance: DC++ (bug fixed)

8

Applications

With nearly any programming paradigm, a wide variety of applications are theoretically possible, since programs that were written with one paradigm can usually be converted to another one, although often with a large change in the level of complexity, and sometimes with a gain or loss of features as a consequence. However, unless there is a large feature gain, this by itself accomplishes very little outside of showing that it is possible, since there is generally no need to rewrite a program that already works. After the program is compiled, it generally makes little difference what techniques were used to construct it. It may affect future maintenance of the program, but that is a different issue. Therefore, in presenting the applications possible with a particular programming paradigm, such as with the Dynamic Encapsulation Model and DC++, the scope should be limited to showing those features that are significant enough to make it worthwhile to rewrite code using the new model.

By developing code with dynamic encapsulation, or more specifically by using the DEM model for constructing type interfaces, it is possible to make programs that are very different after compilation, because many new flexibilities have been added to the run-time system. The code is no longer dependent on any particular implementations for the objects it references, even if some initial implementations were provided in the compiled code. All functions that accept and return instances of abstract types instead of concrete classes are able to adapt to future implementations without requiring recompilation.

8.1 Operating Systems

The operating system provides the interface that all user programs use to communicate with system resources. It also provides the interfaces that device drivers use to provide a consistent resource view to the rest of the system. By using DC++ to write these interfaces, a wide range of new possibilities are available, since all of the interfaces are now much more flexible.

8.1.1 *Remote Method Calls*

Remote method calls are a natural part of the system, and require nothing different from local method calls, in terms of invocation, or in terms of the implementation of the objects. The only difference is that a remote object must have a local object descriptor that acts as a lightweight proxy, by containing a reference to an object manager for the remote object, as well as a handle that the remote object manager can use to locate the object. Note that the proxy objects are

lightweight in the sense that they share a single local object manager that acts as a proxy for a remote object manager. A full-size proxy, including all necessary network connection information, may be needed for the remote object manager, but not for the objects managed by it. In this way, only enough information is needed in each lightweight proxy to uniquely identify objects that share the same remote object manager.

8.1.2 *Kernel Calls*

Kernel calls no longer need to exist as a predefined set of special traps. Instead, the separation between user mode and supervisor mode can be viewed as being separate address spaces, even if the address spaces are actually simultaneously mapped, and a Kernel-Mode Object Manager (KMOM) can be created for kernel calls. These calls can be seen as a specialized form of remote object call, with less overhead since the KMOM is on the same machine.

To optimize performance, implementations of objects that need to run in supervisor mode can flag which methods require supervisor mode to invoke them. This is done by giving each class that might require supervisor privileges a special type, `t_kmode` that has access information as well as a `need_kmode` method that can be invoked with a type and a method index, and will return a boolean value that tells whether the implementation for that method requires kernel (supervisor) mode.

Then, when a user mode thread binds a type to the object, the KMOM will set up an *itable* that will use the Object Manager only to invoke methods that require kernel mode, and will call directly into the object for all other methods. Portions of the kernel address space could be made read-only in order to allow many attributes of kernel objects to be read without entering kernel mode, as long as access to such information does not cause a security violation.

Threads that are running in kernel mode will always access kernel objects directly, never using the KMOM. For this reason, argument validation, in terms of the address ranges specified, should be performed by the KMOM, and not by the objects, unless extra checks are necessary within the kernel as well. The KMOM will validate the addresses of all objects passed in as arguments, and generate proxy objects as needed for accessing them. All method accesses performed on these objects while in kernel mode will go through the proxies, causing the KMOM to be invoked again to force a switch back to user mode for the calls. In this way, arbitrary objects may be passed to kernel-mode functions, but security is not violated if these objects are invoked.

Kernel mode objects must have their class descriptor mapped into memory that can be written to only by the kernel. Attempts to bind an object that claims to be using the KMOM to a class that resides in unprotected memory will fail. Also, calls to kernel-mode methods are never done directly. Each kernel-mode object must have its class registered with the KMOM the first time that class is used, by a thread that is already running in kernel mode. The KMOM builds dynamic tables of valid methods, and every future attempt to invoke a kernel method is routed through these tables. This ensures that no arbitrary addresses in protected or unprotected memory are jumped to while the processor is in kernel mode.

8.1.3 *Boot Flexibility*

The boot process can be greatly simplified through the use of forwarders and object replacement. In microkernel-based architectures, there are many order dependencies, where certain services cannot be initialized before other ones. This especially poses a problem with drivers needed to access the file system, and with basic memory management. As a result, the ‘‘micro’’ kernel often has much functionality added to it just for the sake of initialization, even though it would be preferable to move that functionality elsewhere.

Instead of building all core services into the microkernel, each service can have an object that handles it. For instance, the root filesystem will be managed by a filesystem object. When initially booting, simplified forms of these core objects can be used, whose sole purpose is to be small and to only do what is necessary to get the system up and running. Later, once more extensive objects are able to be loaded for these services, they can be attached to the original objects as forwarders. At such times as the original objects are no longer needed at all, they can be removed and replaced with the new objects. At all times, any handles to the original service objects used at boot-time will stay valid. The actual object these handles refer to will change as the forwarders are added and as the objects are replaced.

As long as proper reference counting schemes are used for keeping track of the bindings, objects that were used only at boot time can be removed from the system when they are no longer needed, and thus memory and other resources can be reclaimed.

8.1.4 *File Systems*

A File System Object Manager (FSOM) can provide persistent storage for objects, thus acting like an object database. By utilizing it in this manner, it is possible to store objects permanently just as easily as they can be passed as arguments. Objects written to the FSOM can be deep or shallow representations, depending on whether the complete state of the object is needed, or only a temporary copy of the first level of it.

An FSOM can sit on top of a traditional file system, using the raw binary files as streams of storage for objects. By also keeping track of which files have their type or class information included, and by having a means of detecting the type of files that do not explicitly specify their types, an object-oriented file system can be blended almost seamlessly with any existing file systems, by providing an interface that adds the required type information to existing untyped files.

8.2 Shell Programming Languages

8.2.1 *c++sh*

Using the `CLex`, `CParse` and `vCAction` classes, it should be possible to write a shell language based on C++ or DC++, which could be called either c++sh or dc++sh, for lack of better names at this time. Essentially, this shell would be a C++ interpreter, with the ability to enter immediate code as well as declare classes and functions to execute later.

If the shell was made to use DC++ syntax in addition to the core C++ language, such a shell would make the full power of the dynamic object system immediately accessible. In a system using a DC++-based operating system, such as the new ShagOS, this shell would be as powerful as any DC++ program that could possibly be written, and it would have the flexibility necessary to immediately test out ideas without waiting for a recompile.

In addition, a dc++sh shell script could be easily converted into standard DC++ and compiled, if it was found to be used commonly enough to warrant the compilation delay and size

increase in return for the faster execution speed.

8.3 Compilers

Compiler writing can be greatly simplified by using the DECO object framework. The job of parsing still needs to be accomplished, of course, but keeping track of datatypes and the layout of structures and classes can be done by simply building dynamic DECO classes to describe the information.

The best advantage of this approach is that classes created in this fashion are not simply representations that can be used for later code generation. They are actually usable classes, of which instance objects can be made, whose methods can later be invoked. This makes it easy to evaluate all code based upon constant values, and have the compiler do as much work at compile-time as possible, rather than generating unnecessary run-time code.

The DECO compiler was originally supposed to use this approach for all of its type management, to greatly simplify the internal works, since the object descriptions it needs to output would be the same objects it uses internally. Unfortunately, due to time constraints, this conversion has not yet taken place. □

8.4 Windowing Systems

Many window management systems are already object-oriented in the way they maintain window characteristics. Most allow flexible attributes to be added to windows, and inherited from parent windows. Rather than having a separate scheme for maintaining the object-oriented nature of a window manager, the DEM object model could be used and then windows would have the same interfaces and functionality as everything else in an object-oriented system based around that model.

When combined with an operating system and shell based on the DEM, this would mean that it would be possible to `cd` (change directory) into the object representing a window, list its attributes, find out its supported methods, and issue commands to it directly (provided that a forwarder enforcing security was not intervening).

8.5 Databases

Object-oriented databases have been a reality for some time now. Using the DEM model it would be possible to have a choice of either directly maintaining database objects with the standard class and instance implementations, or to use a different format internally (such as an already existing database standard) and create an object manager that is in charge of interfacing to the database.

□ Since DECO needs to exist in order to generate the implementations for the objects, it cannot be built using such objects initially.

In this way, objects outside of the database could interact with it seamlessly, and the method invocations would get passed on through the object manager to manipulate the database.

8.6 Expert Systems

Expert systems need a way of modeling the information they're dealing with. The DEM framework provides such a model, since it allows not only objects of actual entities to be created, but also objects that represent the abstract concepts of types, classes, and even meta-types and meta-classes, to whatever degree is necessary. Since it provides a consistent view of all of this information, and a consistent way to manipulate it, this can greatly simplify the job of writing an expert system based on this "knowledge representation."

The flexibility of the system also allows the possibility for new information to be added in the future in a way that does not require recompilation of the code of the expert system that utilizes it. This allows such a system to be updated while it is running.

8.7 Portable Byte-code

The DEM model is constructed in such a way that classes can be implemented in any way feasible on the machine the code runs on. This means that portable byte-code, such as that used by the Java Virtual Machine (JVM) or Inferno's DIS Virtual Machine could conceivably be used for the implementation. This would allow migration of a class to any machine that had the appropriate virtual machine implementation, rather than tying a class to the architecture it was compiled on.

This is possible due to the nature of the *itable*, and the `bind` call. The `bind` can generate stub routines that invoke the virtual machine with the address of the byte-code. It can then put the addresses of these stubs into the itable. Since the code compiled to use a particular type interface looks up the method address from the itable after binding, it will always get the correct implementation.

9

Conclusions

Ian Joyner, in his "Critique of C++ and Programming Language Trends of the 1990s" [*Joyner96*] stated that "bolt-on approaches to object-orientation usually end in disaster." I should have heeded his warning, which was directed primarily at C++. Instead, I attempted to create an additional bolt-on to C++ in order to remedy many of its flaws and produce a more flexible object-oriented programming language that was compatible with C++.

Ian also cites the attempts to be backward compatible with C as being some of the biggest flaws of C++. Likewise, I would have to say that one of the biggest flaws of DC++ is its attempt to be backward compatible with C++. I believe I could have been much more productive creating a new language that directly supported the core ideas of the model without carrying around all the excess baggage of C++.

A few specific examples from his criticism, and how the situation got worse with DC++, were:

- "feature redundancy; too many ways to do the same thing" [*Joyner96*]
  DC++ expands on this even further by giving many implementation options for classes and dynamic classes, and many ways to invoke them. It is possible to invoke implementation classes directly, or to use type interfaces. The data and core functionality of a class can be embedded directly, or can be inherited from other classes.

- "already too large and complicated" [*Joyner96*]
  Parsing C++ was turning into a nightmare. Writing SHACC made things easier, but there were still many complexities dealing with the semantics as well as the syntax. Trying to shoe-horn the DC++ syntax into the C++ language in a way that was consistent, sensible, and did not clash with C++ was quite difficult at times. Implementing the changes in DECO became far more complex than I had originally intended or desired.

- "no uniform way to access constants, variables, and value-returning routines" [*Joyner96*]
  This is one area that only got worse, not better. In a sense, it could be said that a uniform way came out of forcing all accesses through types to look like function calls, but this means that simple field accesses look a lot bulkier than they would otherwise need to be. This is one of the primary areas that I would address in designing a new language. Fields should be accessible with the same syntax as simple variables (i.e. `a.f1 = b.f2`), but should be able to be implemented with functions if necessary. I started to add the ability to auto-generate accessor functions in DECO, but this still would not have allowed C++ calls that look like simple field accesses.

9.1 Problems Encountered and Solved

Needless to say, many problems were encountered during the development of this thesis, and I made an attempt to solve all of them, rather than just documenting them as being problems. This is one of the reasons that writing this thesis and developing the associated code took so long. A few of the more significant problems are discussed below.

9.1.1 *Types vs Classes*

When this thesis work began, the goal was to develop a flexible object-oriented extension to the C++ language that provided *dynamic encapsulation*. That is, I wanted the abstraction of the objects to persist at run-time, so that code would not need to be recompiled to use objects with completely different implementations. Parts of the system should always be exchangeable at run-time, by providing a method-call interface that made no assumptions about the size, layout, or implementation functions of the objects being invoked.

Originally, this was just a way of extending classes so that methods could be dynamically selected, but there were high overheads involved in terms of both code size and execution speed. Also, all type checking was lost, since it was not known until run-time if a particular method actually existed.

The solution to these problems was to break the single hierarchy in C++ into separate *type* and *class* hierarchies. This then allowed the types to be used as interfaces, with proper type checking, and the calls through the types would be handled by the implementation functions of the object that was bound to the typed references.

Switching to this different model was a major breakthrough, but occurred after a lot of work had already been done. Some of the previous work found its way into this paper, especially in the "previous implementation" part of the "Implementation" chapter, but for the most part, I was forced to start over to follow these new concepts.

9.1.2 *Dynamic Encapsulation Model*

All of the work was initially focused on developing the implementation in C++ without much thought of a general-purpose model. I had a goal to be able to interact with other languages, but too much of the design was centered around C++. This often made it difficult to see the big picture, since I was caught up in the details of trying to create an efficient C++ implementation. There was also a problem trying to mentally map parts of the implementation back onto the concepts they represented, and I sometimes found myself going in circles and not really getting anywhere.

At some point, I stood back and realized that I needed to create the conceptual model first, and develop it in a language-independent manner. Then, I could go back to C++ and try to map the concepts onto language features, or extend those features with DC++ where necessary. Thus, the Dynamic Encapsulation Model, or DEM, was born. Quite frankly, it is still in its infancy, and will probably undergo a lot of changes someday if I try to use it to develop a new language that is better suited for it.

9.1.3 *C++ Parsing and SHACC*

One of the major problems encountered with this thesis was the need to correctly parse C++ code. The CParse class was written as a way to isolate the parsing of the input code from the manipulations and code generation that DECO performs, but actually implementing the CParse class turned out to be a major chore.

9.1.3.1 *YACC*

Initially, YACC was used as the parser generator, but YACC suffers from the limitation of only having one token of lookahead. This means that at any given point in the parsing, it can only consider its current state, the current token, and the following token when deciding what action to perform next. To be fair, it does create special states for all the ambiguities within rules, so that as long as no rules have to be reduced, it can essentially follow many paths at once. However, when parsing a big complex language like C++, a clean YACC grammar for it results in many ''shift/reduce'' and ''reduce/reduce'' conflicts, basically meaning that the ambiguities still exist when the ends of some rules are reached, and YACC does not have enough information to pick the right course of action, since it is limited to peeking ahead at only one token.

9.1.3.2 *SHACC*

The solution to this problem was to write a new parser generator that was not limited to a single token of lookahead. The result of this was SHACC, which stands for Shaggy's Homebrew Alternative Compiler Compiler. [⊕] SHACC can parse grammars that are ambiguous all the way to the last token. Of course, if that last token does not make it unambiguous, it will complain. SHACC keeps track of all valid paths through the parse tree simultaneously, adding new paths whenever there are multiple options, and deleting paths whenever the next token is in conflict. Whenever only one path exists, meaning that the parse is now unambiguous at this point, it performs all of the reductions that had been delayed for that path, and executes all of the associated actions.

SHACC takes grammar files that use the same syntax as YACC, with a few rule extensions that have been added to help disambiguate the grammar:

- `%rprec #`
  This applies the given number as a reduction precedence for a rule. If multiple reductions can occur to the same rule, and some of them have precedence values, only the one with the lowest precedence value will be kept.

- `%uniq`
  This tags a reduction that must be unique, meaning that no other active shifts or reductions can be happening at the same time. If the grammar is such that this is not true, an 'ambiguous grammar' error will be reported. This final implicit rule, which shifts and reduces the EOF token, is implicitly tagged as `%uniq`. If a rule is required to be unique, and ambiguities can still result, then the rule or some of its dependencies will have to have `%rprec` reduction precedences applied to them.

[⊕] This is an attempt to keep the name in the spirit of YACC, which stood for Yet Another Compiler Compiler.

SHACC was the most complete project to come out of this thesis, which is ironic since it was just a side effect of the thesis, resulting from the need to write a cleaner C++ parser that was easier to work with than the one written in YACC. A full Unix-style manual page for SHACC may be found on page 193.

9.1.4 *Member Function Pointers in C++*

Since DC++ is written as an extension of C++, it only makes sense that DECO should pass existing C++ code straight through, and should only embellish the parts that are specific to DC++. Specifically, classes that are implementing types should be able to exist as normal C++ classes which have extra methods which tell how to bind the objects to DC++ interface types. Unfortunately, this turned out to be harder than it sounds, since C++ no longer allows the implementation of member functions to be made accessible.

In order to bind an implementation to a type, an *itable* needs to be generated, which has function pointers for all of the methods. In the earlier days of C++, a method was not really that different from a normal function which had an implicit argument called `this`, which was a pointer to the class. With earlier C++ compilers, it was possible to get the address of methods just like any other function. Times have changed, and the newer versions of GCC no longer make this so easy.

C++ and GCC have concepts of method pointers, but these are not at all the same, and are not desirable for implementing DECO. Method pointers carry with them a lot of additional baggage, since they must be able to deal with virtual functions correctly. This results in an increase in both the function pointer size, and the generated code used to invoke the function. None of this was really considered acceptable since the goal was to develop a system that had minimal overhead for simple calls.

As it turned out, old C-style casts still work in GCC to be able to take the address of a member function and convert it to a normal C++ function pointer. With the newest compiler, this gives a warning, but it compiles correctly. However, the new C++ recommended casts `static_cast<>` and `reinterpret_cast<>` do not work at all for this purpose.

One additional problem was the need to disambiguate multiple methods that had the same name, but different signatures. The solution to this was to put an extra cast in first, which casts the address of the function to a member function pointer which has the correct argument types. Then, this is cast to a `void*` which is then put into the itable. In the code for DECO, all of the logic for generating the correct code for this is isolated to a single support function, to make it easy to adapt to C++ compiler changes.

9.1.5 *Memory Management*

A lot of effort was spent trying to solve memory management issues and work them into the design of DC++ and implementation of DECO. The goal was to make the issue of binding and unbinding `TRefs` as transparent as possible, and to eliminate the need to explicitly keep track of objects that were bound, so that they would be freed when no more references to them existed.

Many attempts were made to use special wrapper classes with constructors and destructors to handle all memory management so that issues like data going out of scope (off the stack) while still being referenced would not have to be dealt with. Object handles (`TRefs`) were supposed to take care of all memory management issues, and reference counting for binding was to be taken care of automatically.

This could not easily happen with C++, since the whole language is still based on the C concept of memory management where the programmer has to explicitly deal with all of these issues. The only way to avoid this would be to always allocate everything from the heap and manage it with reference counting and garbage collection.

Unfortunately, this was in direct conflict with the attempt to make the system as low-cost as possible, in terms of both memory and execution speed. There were certainly some objects that were large and complex enough already to justify the extra overhead needed to make such a reference-counting scheme work efficiently, but many of the smaller, simpler objects would become majorly bloated by adding it.

The final solution was to do nothing. C and C++ assume that the programmer will deal with such issues, and so it makes sense that the DC++ extensions should have the same requirements. Basically, what this means is that explicit `bind` and `unbind` calls may use reference counting, if the object's implementation cares to keep track. Otherwise, no such information will be kept. Also, even if the object does keep track, there is no guarantee that the user of the object will actually keep an accurate count of all the references. If extra copies of the reference are made, it is up to the user to explicitly call `bind()` and `unbind()` as necessary to make sure that the object knows how many references to it exist.

9.2 Discrepancies and Shortcomings of the System

This thesis is very incomplete, with respect to its original grandiose goals. There are inconsistencies between the proposed theory of the work, and the described implementation, and even between the described implementation and some of the actual code. * I have tried to repair some of these inconsistencies enough to make the software and this paper useful to others, but every fix uncovers more that needs fixing, and undoubtedly this trend will continue. Therefore the reader will have to keep an eye out for these issues, and take what useful information is to be found, while filtering out that which is no longer relevant.

Some of the more significant lackings of this paper and project are described below.

9.2.1 *Distributed Objects*

The original title of this paper was ''Dynamic Encapsulation of C++ Objects for Distributed Object-Oriented Systems''. For quite some time, the entire object model and the accompanying implementation were developed with the idea that they would be demonstrated as part of a functional distributed object-oriented operating system. For a variety of reasons, this never happened.

* There was much more I could have done and wanted to do with this thesis, but lack of time perpetually stopped me. Constant full time employment, including a couple of demanding jobs with lots of overtime and issues that cluttered my brain, always prevented me from devoting large continuous blocks of time to this thesis. I fought for time whenever I could, but it always came in little bits that were far between, and so every time I would revisit the code or the paper, it would take quite a while for me to get in sync with what I was working on again. As a result, there are many inconsistencies due to the passing of months between the small periods of time that I was able to work on it. There are also many large sections that got written for this thesis but taken out of the final version, since there was not enough time to complete them. Perhaps some of this material will find its way into a paper or two in the future.

However, the second half of the title was not dropped until the final year of work on this thesis. As a result, a lot of the design, and even some of the implementation, is geared towards working in a distributed environment. There are, however, a few more issues that would need to be tackled first.

9.2.1.1 *Universal Identification*

Type binding is done by comparing an identifier for the type with the identifiers for the types an object supports, and then filling in the implementation table appropriately. Currently, the type identifier is a pointer to a descriptor for that type, which is guaranteed to be unique within a single address space.

In a distributed environment, some sort of universally unique identifier needs to be devised for each type. In order to maximize interoperability, it should always be identical for types that are capable of being compatible, even if they were developed independently. However, it must be guaranteed that it is never identical for incompatible types. Any sort of hashed value is incapable of making this guarantee. † Using a large enough number, such as 128 bits, may come close, but that remote chance of an incorrect match is still enough to invalidate the use of this technique.

The best solution that was arrived at was to create an identifier when the type is first devised, but there are still issues about trying to ensure that this is a unique identifier. Also, this prohibits independently developed, but compatible, types from ever matching. Of course, comparing full type signatures is not sufficient. The semantics must also match, and since there is no easy way to specify those, it is probably unlikely that any technique will be able to be devised except one that requires the type to come from a single source and then be distributed to all who use it.

9.2.1.2 *Security Issues*

Security is another major networking problem that needs to be properly solved. Forwarders provide a way to attach security controls to objects, but an assumption is still being made that the object system as a whole is not being compromised. Whenever communication is allowed between systems, there is always a chance that the communication itself has been compromised, and thus the information coming from the remote machine may be invalid, either accidentally, or with malicious intent.

In order to deal with these situations, all remote calls must have their parameters thoroughly checked, and this creates an efficiency bottleneck. A lot of effort was put into trying to create an efficient remote object invocation scheme that would not allow a machine to be compromised. Unfortunately, all of the schemes devised either have a high per-call cost, due to the need to verify pointers, or a high object-maintenance cost and potential cleanup problems, due to the need to register objects in indexable arrays of object pointers.

In the current implementation, no security concerns are addressed. This is not quite as bad as it sounds, since remote object invocation does not occur either.

† An exception to this would be a so-called ''perfect'' hash function. However, I do not believe it is possible to create a perfect hash function for all of the potential type descriptors that could be created in the future.

9.2.2 *C++ Interaction*

The interface of DC++ with C++ turned out to be less than ideal. In fact, an extensive amount of time was consumed on this project in an attempt to come up with a clean mingling of C++ and DC++, when it may in fact have been better to just provide a C-style interface between the two languages. Once again, this was a case of mistakenly assuming that because C++ was "object-oriented", it should be possible to build a more extensive object-oriented language around it, in a compatible fashion. Unfortunately, the limitations of C++ turned out to be more prohibitive of this than was originally expected.

A language that already had a clean separation of the concepts of *type* and *class* would have helped significantly, even if it only supported static encapsulation. The fact that C++ only had a single class hierarchy forced me to find ways to build a simulated dual hierarchy of types and classes that would still exploit the language's inheritance properties. This turned out to be rather difficult, and I was always forced to make compromises that would prevent some manner of C++ code from being seamlessly linked with DC++ code.

9.2.3 *Excessive Overhead*

Objects are a human way of modeling concepts. Although objects can be modeled in an executable form on computers, they are far from being the most efficient way of executing software, both in terms of memory requirements and speed. Objects provide a convenient framework for describing a problem in a way that can be understood and modified easily.

Converting one object model (DC++) into another one (C++) does not eliminate all of the overhead, and in fact it loses some important information that could be used to optimize the code. The final optimal code might have no obvious concepts of the objects left, except in places where such concepts are required for interfacing. Even those interfaces could be optimized if they are machine-to-machine interfaces. Machine-to-human interfaces require some sort of translation that allows humans to still see the object view. Unfortunately, allowing this view of the whole system might require keeping around excessive amounts of information that the system does not normally need to run. Therefore, this should just be a debugging option.

Once enough debugging has been done, and a full-speed and small version of the code is needed, it should be compiled without run-time object support, and thus dynamic interfaces should get used only where absolutely necessary. Lots of things get called directly, or inlined and reduced, and a lot of garbage collection or reference counting can be converted to a simpler memory management scheme, perhaps even just a stack, or static allocations.

The object model is great for rapid prototyping-- for getting a system up and running fast, for rebuilding from existing components, and for making changes easily and understanding the impact of those changes. It is too bulky for creating efficient systems. Certainly, there are many ways it can be greatly streamlined, but no single technique offers the full functionality in a way that is even close to optimal. Procedural systems always win out, because they can be hand-optimized to be far more efficient. However, object-oriented systems can potentially be converted to optimal procedural systems, emulating only the OO features that are necessary in a particular case.

Eliminating the run-time object constructs from a system becomes a lot harder, though, when a program interacts with remote machines that can manipulate its objects. Therefore, for distributed systems, it may still be necessary to have the overhead of the object model for the portions of the system that must interact remotely with independently-compiled systems.

9.3 In Retrospect (Lessons Learned)

9.3.1 *C++ Parsing*

If I were to do this project over again, I would just create a new language, based loosely around C syntax, in order to implement my object model. In fact, I may just go ahead and do that anyway in the future. C++ turned out to be much more complex than what I needed to demonstrate my model, which would have been fine if there was some way to just use the simple parts and avoid the complex ones. Unfortunately, every time I thought I could ignore some piece, it later turned out that I needed it.

I had originally chosen C++ because I thought I could implement all of my extensions in a simple manner within the language. When that was unsuccessful, I tried to just pre-process the dynamic classes and types and still use regular C++ for everything else. Preprocessing quickly turned into requiring full lexical analysis and parsing of the C++ language, plus my extensions. In retrospect, at that point I should have reconsidered using C++ and instead devised a language that was easy to parse and just contained the features I needed. Instead, I tried to make it as compatible as possible with existing C++ code so that it could be seen as a true language extension. I looked at trying to use GCC as the base for my C++ parsing, by just adding extensions to it, but after spending a lot of time examining the large and complex GCC source code, I decided it was best to start from scratch and try to parse C++ myself. Some of the reasons for this were:

- Too many extensions had already been shoe-horned into GCC
- The GCC source was immense, and the changes I needed could not be centralized well in it
- GCC's C++ parser already incorrectly parses C++ code that appears somewhat ambiguous but is perfectly valid according to the C++ standard. [‡] I wanted to write a parser that did the job right.
- I wanted to be able to freely release my work under my own terms, and not have it tied to the terms of the GPL (GNU Public License).
- I would have liked to someday turn DECO, or some descendent of it, into a full experimental compiler, and thus wanted my own code base to start from

[‡] As an example, try the following code with GCC:

```
struct f { f(int); f(f&, int); };
int main(int argc, char **argv) { f g(f(argc), 3); }
```

GCC will decide, incorrectly, that the statement in `main()` is a function declaration after it sees the `f(argc)`. Then it sees the 3 and does not know what to do with it so it fails. DECO will realize that this is still ambiguous until it sees the 3 and then it will decide, correctly, that this is a constructor call.

Parsing the C++ language exposed me to all of the major ambiguities in the syntax of C++, and forced me to find ways to deal with them. Eventually, I had to give up on using YACC altogether if I wanted my grammar to stay comprehensible, since all of the workarounds for the one-token-lookahead problem were making the grammar exceedingly complex. At the point when I had to start writing SHACC as a YACC replacement, I *really* should have considered implementing a new language instead of trying to parse C++. [¤]

The bottom line is that C++ is too large and complex of a language to attempt parsing for any research project, unless of course the focus of the project is parsing C++. Any and all alternatives should be actively sought out, such as inventing an alternative language, or using the facilities of existing parsers, such as parts of C++ compilers, or even the result of my efforts. For this reason, among others, I am making my C++ parsing source code available so that others who would consider parsing C++ may benefit from my work and not have to start from scratch like I did.

9.3.2 *Backups and Source Control*

Nearly everyone knows how important backups are, and most people realize the importance of using a source control system. Unfortunately, I was in the habit of only using a source control system for actual source code, not for documents. As the result of a thoughtless mistake [◊] I lost around fifteen minutes of work, but in my attempt to recover it I accidentally overwrote the only copy of the newer file with a much older editor backup and ended up losing several days worth of work, consisting of about 12 pages. About 2 pages of information was recovered from hunting through the raw data on the disk, but much time was lost regardless.

If I had been using source control for the documentation of my thesis, and regularly checking in significant changes, this sort of loss would likely not have happened. As a result I am now using source control for everything, [¥] including documents and source code, and I am also trying to get in the habit of performing backups more frequently.

9.4 Future Research

The research and development performed during the writing of this thesis has shown a variety of areas that could be further explored and developed in the future, including the following:

- **Dynamic C++**
  Some of the concepts of extending C++ could be better integrated with the language, to become proper language extensions to propose for a future standard. Or alternatively, a new language could be developed around the core features presented here, eliminating

[¤] Or perhaps I should have proposed a new thesis, making SHACC the primary focus and saving DECO and DC++ for future work.

[◊] Never start X-Windows on a laptop computer from a remote login session, and then continue work in that X-Windows session from elsewhere after the network is disconnected. Eventually, many hours later, the login will time out and take X-Windows down with it. I learned this the hard way.

[¥] I started out just using RCS with some custom scripts to manage the project as a whole, and now I'm using CVS (which calls RCS) to manage my source tree. I highly recommend CVS to anyone looking for a source control system, especially since it is free.

  other language features that are unnecessary.

- **New Languages**
  Rather than extending C++, a new programming language could be developed whose core is based around the separation of the concepts of types and classes. Such a language would not have to carry around all the excess baggage of C++, although C++ code could still be interfaced with it by using DECO to make C++ code conform to the type interfaces.

- **Virtual Parsing**
  The ideas used by `CParse` to separate the parser from the actions to be performed could be taken a step further, to form a standard interface between parsers and interchangeable actions. See *Section C (DECO Implementation)*, starting on page 139, and specifically page 141, for more information about the virtual abstraction used for actions performed during parsing.

- **More Efficient Parser**
  To handle the complexities and ambiguities of the C++ language, SHACC was developed as a more-powerful alternative to YACC. Unfortunately, SHACC does most of its work at run-time, consuming much time and memory. Taking SHACC to the next level would be an interesting project, by having the SHACC compiler do much more extensive analysis of the grammar, and hopefully just build some Pushdown Automata (PDAs) that can be used to do the parsing with the minimal amount of work.

- **Object Transmission**
  Providing ways of transferring objects between machines is a big issue. Either a compatible format needs to be decided on in advance for each type of object, or usable information needs to be provided with the object describing it. Decisions also need to be made about whether the object should be transferred or accessed remotely via a proxy, whether all or part of an object should be sent, whether some parts should be cached, and how to maintain coherency, etc.

- **Meta-data**
  Much more research can still be done in the area of meta-data or data whose purpose is to provide higher-level information about the structure of other data. Providing such information in a standard manner is crucial to making flexible systems that are truly portable.

- **Virtual Machines**
  One way of making data portable is to provide meta-data in a form that is actually code rather than tables of information. By using virtual machines, it is possible to run that code anywhere, and this technique may turn out to be a lot more flexible than providing tables of meta-data. However, it can also lead to security problems, and seems to require a more complex implementation.

- **Object-Oriented Operating Systems**
  The original intent of DC++ was to provide the tools necessary to continue work on the new ShagOS. Since so much time was spent on the tools, the operating system development has stagnated. Much future research can be done on utilizing these tools to develop an object-oriented operating system, whether it be ShagOS, or some entirely new project based on different ideas.

- **Standardized Type Interfaces**
  For systems based on types to be useful, standard sets of types need to be universally

agreed upon. Developing sets of such types (independent from implementations), with naming conventions and semantics that are short and concise yet fairly free of ambiguity could take extensive research. Actually getting people to agree upon such conventions, however, could require a lot more than just research.

- **Standard Algorithm Implementations**
  Many well-known algorithms could be implemented in C++ in a way that does not require future recompilation. In this sense, they could be used to build libraries of useful routines, and could provide much canned functionality that is often re-implemented otherwise. This is in many ways similar to what the Standard Template Library provides, except that the STL does not separate the interface from the implementation at run-time, and thus code using the STL usually requires recompilation of everything whenever changes are made to the implementations.

With the multitude of ideas that were explored or at least considered while working on this thesis, it is my hope that I will be able to build upon the experience gained, and hopefully the code as well, and may explore one or more of these other areas in the future, perhaps as part of a PhD thesis.

A

DC++ Grammar Summary

The syntax for DECO is expressed below as an extension to the existing C++ syntax, utilizing the same syntax notation used in *The C++ Programming Language* [*Stroustrup93*] [*Stroustrup97*] [@] Starting with the C++ language as defined in the latest ISO Draft Standard for the C++ language, [*Koenig96*] the following changes need to be made to the existing language to support DC++:

[@] This method of expressing the grammar was chosen, instead of the standard BNF (Backus-Naur Form), to remain consistent with other standard specifications of the C++ programming language.

A.1 Classes

These changes are needed to allow the declaration of types, dynamic types, and dynamic classes:

class-specifier:
  class-head `{` *member-specification*$_{opt}$ `}`
  class-head `virtual` `{` *member-specification*$_{opt}$ `}`
  class-head `dynamic` `{` *member-specification*$_{opt}$ `}`
  class-head `dynamic virtual` `{` *member-specification*$_{opt}$ `}`

member-specification:
  member-declaration *member-specification*$_{opt}$
  type-access-specifier **:** *member-specification*$_{opt}$
  access-specifier **:** *member-specification*$_{opt}$

member-declaration:
  decl-specifier-seq$_{opt}$ *member_declarator_list*$_{opt}$
  function-definition **;**$_{opt}$
  qualified-id **;**
  using-declaration
  template-declaration

type-access-specifier:
  type-identifier
  `dynamic` *type-identifier*

A.2 Declarators

To support conditional dynamic binding, the `?=` form needs to be added for initialization:

initializer:
  `=` *assignment-expression*
  `?=` *assignment-expression*
  `=` `{` *initializer-list* **,**$_{opt}$ `}`
  `(` *expression-list* `)`

A.3 Expressions

For allowing conditional dynamic assignment, the `?=` operator needs to be added:

assignment-operator: one of

```
=   ?=   *=   /=   %=   +=   -=
>>=   <<=   &=   ^=   |=
```

A.4 Overloading

In addition, to support conditional assignment and binding in an orthogonal manner, and allow the implementations for dynamic classes and object references to be defined using the language, rather than in the compiler, the following need to be added to allow defining the constructors, assignment operators, and conversion operators for conditional assignments and conversions.

operator: one of

```
new   delete   new[]   delete[]
+   -   *   /   %   ^   &   |   ~
!   =   ?=   <   >   +=   -=   *=   /=   %=
^=   &=   |=   <<   >>   >>=   <<=   ==   !=
<=   >=   &&   ||   ++   --   ,   ->*   ->
()   []
```

conversion-function-id:

`operator` *conversion-type-id*

`operator ?` *conversion-type-id*

B

DECO Manual Page

NAME

deco − DC++ Preprocessor and Compiler (v0.8)

SYNOPSIS

deco [**−x** *language*] [*cpp-options*] [**−E**] [**−S**] [**−mc**] [**−mcc**] [**−mh**] [**−c**] [**−np**] [**−v**] [**−w**] [**−W** *warning*] [**−fsyntax-only**] [**−fsavecode**] [**−finclevel=***num*] [**−frealtypes**] [**−ffullscope**] [**−fwidth=***num*] [**−finterface**] [**−fonlyinterface**] [**−g**] [**−dM**] [**−dN**] [**−dD**] [**−dX**] [**−dy**] [**−dA**] [**−dT**] [**−dC**] [**−dE**] [**−da**] [**−ddef**] [**−ddecl**] [**−save-temps**] [**−oh** *hdroutfile*] [**−o** *outfile*] [*infile*|**−**]

DESCRIPTION

DECO *(Dynamic Encapsulator of C++ Objects)* is a pre-processor and compiler for DC++ code, which is a language that extends C++ with dynamic types and classes. **DECO** can be used for several different purposes:

- Header file pre-processing; DECO can convert DC++ header files (.dh files) into a standard C++ header (.dh.h) and a code file for dynamic type and class information (.dh.cc).
- Compiling; DECO can compile DC++ programs into C++ or C code, which can either be output, or run through the native C++ or C compiler to produce an object file.
  Note: This latter ability is not currently fully functional, and probably should not be used until a future release.

OPTIONS

Options may be supplied in any order, although options should precede any filenames they affect, or other options that they may interact with. For instance, the −x option, for overriding the language of a source file, must precede the source file specifications it refers to.

The following is a quick summary of the options-- full explanations can be found in subsequent sections.

Overall Options
−x *language* −S −E −mcc −mc −mh −c −np −oh *hdroutfile* −o *outfile* −v

Preprocessor Options
−E −i*option file* −nostdinc −nostdinc++ −nostdincd++ −undef −C −P −H −A*command* −D*macro*[=*defn*] −U*macro* −dM −dN −dD −M*command* −I− −I*path*

Language and Parsing Options
−fsavecode −finclevel=*num*

Formatting Options
−ffullscope −frealtypes −fwidth=*num* −finterface −fonlyinterface

Warning Options
−w −fsyntax−only −Wall −Werror

Debugging Options
−g −dM −dN −dD −dX −dy −dA −dT −dC −dE −da −ddef −ddecl −save−temps

Code Generation Options
(not yet supported)

Optimization Options
(not yet supported)

OVERALL OPTIONS

−x *language*
Specify the language of source files that follow. This can be **dc++** for DC++ source files to compile, **dc++hdr** for DC++ header files to preprocess, or one of **c++**, **c++hdr**, **c**, or **c-hdr**.

−S Stop after the compilation stage, and leave the assembler output. This feature is not currently implemented.

−E Stop after processing with **cpp**, and send the preprocessed output to the standard output.

−mcc Stop after converting from DC++ to standard C++, and leave the C++ output in a ‘.cc’ file.

−mc Stop after converting from DC++ to standard C, and leave the C output in a ‘.c’ file. In the future, either **−mcc** or **−mc** will be supported for any particular implementation, but probably not both.

−mh Generate a standard header file for the following code, by tacking the .h extension onto the basename of each source file. This is the same as **-oh .h**.

−c Stop after compiling the output into a ‘.o’ file. This option is not needed when preprocessing header files, but when compiling DC++ source files, the current version of DECO will complain if this option, or one of the previous ‘Stop’ options, is not specified, since the final linking stage is not yet supported.

−np Don’t preprocess. This causes the C preprocessor to be bypassed.

−oh *hdroutfile*
Output a standard header for the current code to the specified output file. The **-mh** option is automatically implied by this option. If the output filename begins with a dot (period), then the output filename for headers is formed by tacking the specified extension onto the basename for each source file.

−o *file* Put the final output into the specified file, overriding the defaults for output file naming.

−v Print version information about DECO, and print the commands for each substage of the compilation process, along with version information for each invoked program.

PREPROCESSOR OPTIONS

With the exception of -E these options are not currently handled internally by DECO, but are instead passed on to the standard C/C++ preprocessor (cpp). Therefore, not all of these options will necessarily work as expected. (In fact, in the current version, they’re not implemented at all, so don’t even bother trying... just preprocess the output separately with cpp first)

It should also be noted that the following definitions are implicitly passed on to the pre-processor prior to all others:

-D__asm__=asm -D__const=const -D__signed__=signed -D__inline__=inline

−E Execute just the preprocessor stage.

−i*option file*
Specify include file options, such as '-include', '-imacros', '-iprefix', etc. Basically, any option that starts with '-i' will be passed on verbatim along with the following parameter to the preprocessor.

−nostdinc
Don't search the standard directories for C header files.

−nostdinc++
Don't search the standard directories for C++ header files.

−nostdincd++
Don't search the standard directories for DC++ header files.

−undef Don't predefine macros.

−C Don't discard comments.

−P Don't generate #line commands.

−H Print the name of each header file used.

−A*question_or_answer*
Assert the answer for a question, or disable standard assertions. (The precise meaning of this can be found in the **cpp(1)** manual page, if it is supported)

−D*macro*
Predefine a macro.

−D*macro***=***defn*
Predefine a macro with the specified definition.

−U*macro*
Undefine a predefined macro.

−dM List all macro definitions after preprocessing.

−dD Include all macro definitions in the output.

−dX Show all command invocations that are executed, such as the command line used to call the preprocessor.

−dN Include all macro definitions in the output, but with just their names.

−M*command*
Specify commands for 'make' dependency rules, such as '-M', '-MM', and '-MG'.

−I*path* Add another path to the list to search for local include files.

−I− Specify that the following '-I' commands affect system includes as well as local includes.

LANGUAGE AND PARSING OPTIONS

−fsavecode
Save the tokenized form of all code, rather than parsing it. This might help with some programs whose code structure is not properly parsed by deco. The output will be a somewhat formatted version of the token stream for each block of code, with no interpretation.

−fincleve1=*num*
Set the include level at which real headers begin. This defaults to 0, which means that all `#include`'d files are considered headers, and the declarations and definitions contained in them are tagged as being from a header. By setting this to a higher number, one or more levels of included files can be considered part of the main body of code, rather than being from a separate header. This is mainly useful when preprocessing .dh (dynamic

header) files, since this allows multiple levels of included files to considered part of the code to be converted and sent to the output.

FORMATTING OPTIONS

−ffullscope
Print the full scope of all identifiers in the output, as well as in warning and error messages.

−frealtypes
In the output, and in warnings and errors, print the real type of all declarations and casts (and wherever else a type might be printed). All types which contain a reference to a typedef'ed name will now have the actual complete type substituted in its place.

This can be useful for taking multiple levels of typedefs and seeing what the single declaration for the new compound type would be. This can also be useful when debugging to see exactly what the final declared type of some entity is.

−fwidth=*num*
Set the output width to *num*. This sets the point at which line breaks are inserted in the code output. The continuation line is justified to one character greater than the current indentation level.

−finterface
Output a useable interface, with no unnecessary implementation. This means that static private data and private functions will be removed, along with static global data.

−fonlyinterface
Output only the interface, and none of the implementation. The resulting code is for exposition only, and probably cannot be compiled as-is, since it may be missing pieces such as inline functions and private data members that affect the size of structures.

WARNING OPTIONS

−w Turn off all warnings.

−fsyntax−only
Check for syntax errors, but produce no output.

−Wall Turn on all sensible warnings.

−Werror
Consider warnings to be errors.

DEBUGGING OPTIONS

−g Produce debugging information in the compiled output. When preprocessing header files, this will generate extra debugging information in the corresponding '.dh.cc' file.

−dM List all macro definitions after preprocessing. Currently, this is just passed on to the preprocessor.

−dD Include all macro definitions in the output. Currently, this is just passed on to the preprocessor.

−dN Include all macro definitions in the output, but with just their names. Currently, this is just passed on to the preprocessor.

−dy Dump debugging info during parsing.

−dA Dump actions generated by parser.

−dC Turn on additional parser debugging.

−dT Show internal (nested structure) representation of types. This affects both warnings/errors and the final output, so this will make the output unable to be compiled, and only useful for debugging, or for finding out the true meaning of a convoluted C/C++ declaration.

−dE Print all expressions in an indented tree format prior to printing them in standard infix notation.

−da Produce all debugging output. This is the equivalent of all of the above -d* options.

−ddef Print the location of all symbol definitions.

−ddecl Print the location of all symbol declarations.

−save-temps
Output all temporary files.

CODE GENERATION OPTIONS

These are not yet supported, since the full compiler portion of DECO is not yet fully implemented.

OPTIMIZATION OPTIONS

These are not yet supported, since the full compiler portion of DECO is not yet fully implemented.

PREDEFINED MACROS

The following macros can be predefined by DECO:

__dcplusplus
This is always defined, and can be used to check whether the source is being run through DECO (or another DC++ compiler).

__dcplusplus_hdr
This is defined when a header file is being preprocessed, and can be used for conditional compilation based on whether the '.dh' header file is being used to generate a '.dh.h' file, or whether it is simply being included from a '.dc' source file which is being compiled.

DECO SYNTAX EXTENSIONS

DECO input consists of standard C++ augmented with dynamic classes and types.

A dynamic class can be declared with 'dynamic class { }'. It is common practice to include a header file which contains the definition `#define dclass dynamic class`, allowing a dynamic class to be specified simply with 'dclass {}'.

A dynamic type can be declared with 'dynamic virtual class'. It is common practice to include a header file which contains the definition `#define dtype dynamic virtual class`, allowing a dynamic type to be specified simply with 'dtype {}'.

Both dynamic classes and types can specify sections which pertain to specific types, by using 'typename:' inside the aggregate definition. This follows the same syntax as 'private:', 'public:', etc, except that instead of merely defining a scope, it also builds dynamic access information to be bound to the specified type.

For a full description of the use and features of dynamic types and classes, see the thesis *Dynamic Encapsulation of C++ Objects for Distributed Object-Oriented Systems* which is available from *http://www.csh.rit.edu/˜shaggy/thesis.html*

DECO SOURCE REQUIREMENTS

Any source files using dynamic types and classes must include the standard DC++ object header, as follows:

```
#include <dcobj>
```

This file contains the definitions for an object, and all the supportive functionality, as well as macros such as 'dtype' and 'dclass'.

Since the code generated for dynamic types and classes requires many external definitions, an error will be reported if this file is not included.

INPUT FILES

file.h C/C++ header file
file.dcc DC++ source file

file.dh DC++ header file

OUTPUT FILES

file.c C source file
file.cc C++ source file
file.ii preprocessed C++ source file
file.s assembly
file.o object file
file.dh.h C/C++ header (from dynamic header)
(alternative: file.dhh)
file.dh.cc C++ source (from dynamic header)
(alternative: file.dhc)

OTHER FILES

TMPDIR/dcc* temporary files used during processing

TMPDIR comes from the environment variable, or defaults to /tmp.

SHORTCOMINGS

At the moment, it can fully parse DC++ code, but can't generate all of the automated code. This is a work-in-progress.

BUGS

Many. (although most are really just incomplete features) And they're probably being worked on even as you read this.

SOURCE

This program, along with its full source code, is available from *http://www.csh.rit.edu/shaggy/software.html* and *ftp://ftp.csh.rit.edu/pub/members/shaggy/shagware*.

SEE ALSO

cpp(1), **gcc(1)**.

AUTHOR

DECO was written by Frank Barrus <*shaggy@csh.rit.edu*>, based on research documented in his R.I.T. Masters Thesis, *Dynamic Encapsulation of C++ Objects for Distributed Object-Oriented Systems.*

C

DECO Implementation

This section describes the classes that are used to create the DECO compiler. *Figure 6-3* on page 83 shows the interaction of these components in DECO.

C.1 CLex

`CLex` is a C++ class that encapsulates the lexical analysis of C, C++, and DC++, with built-in CPP support. [#] It is based around a state machine and is designed to be very fast as well as flexible. It handles recognition and conversion to internal form of all the standard constant datatypes, tokenizing of all operators and keywords, processing CPP directives, and it can keep enough information to be able to regenerate C++ code from tokens, including indentation levels, and line breaks.

C.2 StrTab

`StrTab` is a simple default implementation of a string table for use with `CParse`. In its current implementation it just uniquely stores each new string, and uses the address as the unique identifier. This allows future string comparisons to be made by just comparing pointers for equality.

The `CParse`, `CAction`, and `COut` classes all use a `vStrTab` object for their string table. This is an abstract base class from which `StrTab` is derived. This allows implementations to easily substitute alternative string classes.

[#] The code needed to make use of this is not yet implemented, so a separate pass through a standard C/C++ preprocessor is currently necessary. DECO automatically invokes the standard C/C++ preprocessor prior to sending the input through `CLex`.

C.3 SymTab

The `SymTab` class is a default symbol table implementation for use with `CParse`, just to supply enough of a framework for simple C++ parsing. This can be overridden with customized symbol tables, since `CParse`, `CAction`, and `COut` all use an abstract base class called `vSymTab` from which `SymTab` is derived.

`SymTab` uses a linked-list variation to maintain an ordered set of string identifiers (`vStrTab::Id`) that are in each symbol table. The order is based upon the original order in which entries were added to the symbol table, to allow for the output of symbols in the order they were originally defined. A second set of pointers is used to maintain a cache of sorts that allows the most recently used entries to be accessed the fastest. Every time an entry is looked up in the symbol table, it becomes the new head of the cache list. Since searches always begin from the head of the cache list, the last accessed item will always be checked immediately, and the rest of the items in the list will be ordered according to how recently they were accessed.

C.4 CParse

`CParse` is C++ class that encapsulates the parsing of C, C++, and DC++, and provides an interface that allows it to be used for a variety of applications, such as pre-processors, interpreters, compilers, or source code analyzers. `CParse` is based around a SHACC-generated parser, and uses `CLex` for the lexical analysis. It gains its flexibility through the use of virtual functions for all actions, so that no assumptions are made in the parser about the purpose of the parsed information. This allows the parser to be written once, and heavily optimized and debugged, while being used for numerous different purposes.

The parser used by earlier versions of `CParse` was generated by YACC. Internally, rather than using backtracking, or recursive-descent parsers, as some other C++ parsers do, `CParse` originally used YACC directly, resolving ambiguities by parsing them with separate rules for ambiguities that saved up the actions and tokens on a separate stack. Once some unambiguous point in the grammar was reached, the correct form for each saved action was chosen, and the actions were executed. In contrast, non-ambiguous expressions had their actions applied immediately. Although this technique worked, as the grammar grew larger, and more overlapping ambiguities resulted, the special rules for ambiguities kept getting more and more complicated, and thus error-prone as well.

YACC was eventually replaced with SHACC (Shaggy's Homebrew Alternative Compiler Compiler) instead, and the grammar was greatly simplified. SHACC is a parser generator that takes `.y` files that look much like YACC files. However, unlike YACC, it is not limited to a single token of lookahead, and it has reduction precedence rules to resolve complex ambiguities. The manual page for SHACC, which describes its usage and syntax, can be found near the end of this thesis, on page 193.

C.5 CAction

The `CAction` class is a derived implementation of the abstract base class `vCAction`. `vCAction` provides all of the interface functions to actions needed by `CParse`. The default `CAction` implementation handles C[□], C++, and DC++.

The main purpose of the `vCAction` interface is to separate the work of the actual parsing from the actions that must be performed during parsing. Traditionally, the parser directly invokes all required actions; by separating the parsing and the actions in this manner, it is possible to reuse the same parser with multiple action implementations, for instance some that are tuned towards interpretation rather than compilation, or some that are just for code analysis. In addition, the parser code stays much simpler and easier to maintain.

The `CAction` class implements the majority of the C++ and DC++ semantics and is responsible for maintaining the information about declarations and code sequences. The following are all done by `CAction`:

- building type lists
- converting type lists to a new distinct type
- type checking
- argument/parameter matching
- most error and warning reporting
- type coercions
- constant arithmetic evaluation
- building expressions
- building expression and statement lists
- constructing aggregates
- handling default parameter values
- invoking constructors and conversion operators implicitly
- access rules
- inheritance
- symbol lookup

Since all of the functions that handle the actions are virtual, even when `CAction` is used to implement `vCAction`, any of the actions can still be overridden with customized implementations. DECO takes advantage of this to override such actions as EndAggr so that part of the conversion can take place as soon as a new dynamic class is done being defined.

[□] Full standard C support is currently lacking. In the ways in which C is mostly like C++, it should work fine, but in certain ways that differ, such as scoping rules, the C++ rules will take effect instead of C. This can be fixed in the future.

C.6 Internal Representation

The parsed structure of the DC++ program is stored internally with a combination of many smaller data structures and classes, including `Decl`, `Expr`, `Stmt`, `Scope`. There are also a bunch of object representations of intrinsic C++ types, such as pointers, integers, arrays, and aggregates.

C.6.1 *Decl*

`Decl` is the basic data structure used to store all declarations. `Decl` structures keep all information about their associated type, identifier, and symbol table, and any flags associated with the declaration. A shortened form of `Decl`, known as `TDecl`, is used temporarily during parsing to maintain this information. Type lists used while building up type information are maintained with a `TDecl`.

C.6.2 *Expr and Stmt*

`Expr` and `Stmt` are slightly different variations of an encapsulated pointer to a `ExprNode` structure, which is used to keep one node of an expression or statement. Each node can keep one piece of information, such as a literal or a reference to a declared variable, or it can be used for an operator and up to three pointers to other nodes. Expression nodes can also be chained to make expression lists or statement lists.

C.6.3 *Scope*

The `Scope` class is a wrapper for a `vSymTab` pointer that provides a type-checked interface for utilizing symbol tables as well as creating new ones. Since `vSymTab` is general-purpose, it only uses a `void*` for the associated value of each symbol table entry. The `Scope` class provides a view of symbol table entries with `Decl` pointers.

C.6.4 *CObj objects*

All of the intrinsic C++ datatypes are represented by classes from `CObj`, a collection of objects all derived from one common base-type, and whose dynamic type can be checked. At some point, the DEM model and DC++ are supposed to be used to provide this object collection, but during development they all needed to be hand-coded.

Implementations of the following types are all supported:

— `Integral` (any size signed or unsigned)

— `PtrTo` (pointer to)

— `RefTo` (reference to)

— `Qualifier` (const/volatile)

— `Array` (single dimension; can be nested)

— `Params` (used by `Func`)

— `Func` (non-member function, but could be scoped)

— `Method` (member-function; uses `Func`)

— `Aggr` (struct, class, union, type, dclass, dtype}

The exact datatypes used by a C++ program are constructed instances of these classes, which can be seen as "type templates." For example, the `Array` class can be used to create an object that represents an array of size 20 of integers. In order to do so, an `Integral` instance also needs to be constructed with the correct size and sign information for the integers needed.

Instances of the basic integral types are already included in `CObj`, as follows:

— `Type_char` (8 bits, signed)

— `Type_schar` (8 bits, signed)

— `Type_int` (32 bits, signed)

— `Type_short` (16 bits, signed)

— `Type_long` (32 bits, signed)

— `Type_long2` (64 bits, signed)

— `Type_uchar` (8 bits, unsigned)

— `Type_uint` (32 bits, unsigned)

— `Type_ushort` (16 bits, unsigned)

— `Type_ulong` (32 bits, unsigned)

— `Type_ulong2` (64 bits, unsigned)

C.7 Deco

The `Deco` class is derived from the `CAction` class and thus acts as an implementation of `vCAction`. It extends the functionality of the `EndAggr` action to support the DC++ extensions and also contains all of the code for converting DC++ to standard C++.

C.8 COut

The `COut` class takes the internal representation of a C, C++, or DC++ program and converts it back into text that can be parsed by the appropriate compiler. It will attempt to reuse the original ordering as much as possible, [⊕] but dependency checking is also done and code will be reordered as necessary. This is mainly so that automatic code converters, such as DECO, do not have to

[⊕] There is a `next` field in the `Decl` structures that is used to maintain an ordering of the declarations based on their first declaration or definition.

worry about ordering. They can just construct the appropriate declarations and code sequences, and let `COut` make the output correct. However, since `CAction` is lax about definition ordering, it is possible to input code like the following into DECO:

```
class a;
class b { a x; };
class a { int y; };
```

The problem most C++ compilers have with the above code is that the definition for `class a` is not available when `b::x` is defined. This will get successfully parsed by DECO, however, and will output the following code:

```
// types:
class a {
    int y;
};

class b {
    a x;
};
```

Note that the dependency checking in `COut` automatically reordered the definitions correctly. In cases where it is necessary, forward class declarations will get emitted as well.

It should also be noted that a comment was added before the type declaration. `COut` tries, whenever possible, to output constants, then types, followed by variable and function import and export declarations, followed by actual definitions. All of these sections have a short comment added, and when header files are parsed, an extra comment section is added at the bottom to show external dependencies.

To some extent, `COut` can be seen as a form of C++ pretty-printer, and some of its output formatting options, such as a maximum line-width, and whether typedefs should be expanded where referenced, are configurable.

D

Library Interfaces

Each of the following library interfaces shows only a portion of the complete header file. All of the header files have been run through DECO with the `-fonlyinterface` option which causes the output to omit any definitions or private members of any classes or structures. Thus, the headers shown are useful only for reference purposes and the original headers in the source tree must be used for all real code.

D.1 *vStrTab*

```
// this file was auto-generated by deco 0.12 from the command:
//  deco -fonlyinterface -fsavecode -include stdhdr.hh vStrTab-mi.hh

// types:
class vStrTab {
 public:
    struct _Id;
    typedef _Id *Id;
    inline virtual ~vStrTab();
    virtual Id id(csz *string) = 0;
    virtual csz *string(Id id) const = 0;
    virtual uint ref(Id id) = 0;
    virtual uint unref(Id id) = 0;
};
typedef const vStrTab &vStrTab_;

// external declaration requirements:
//  typedef unsigned int uint;
//  typedef const char csz;
```

D.2 *vSymTab*

```
// this file was auto-generated by deco 0.12 from the command:
//  deco -fonlyinterface -fsavecode -include stdhdr.hh vSymTab-mi.hh

// types:
class vSymTab {
 public:
    typedef vStrTab::Id Id;
    typedef void *Val;
    inline virtual ~vSymTab();
```



```
    virtual bool get(Id id, Val &val) const = 0;
    virtual bool set(Id id, Val val) = 0;
    virtual bool add(Id id, Val val) = 0;
    virtual bool del(Id id) = 0;
    virtual vSymTab *parent() const = 0;
    virtual bool parent(vSymTab *) = 0;
    virtual void *data() const = 0;
    virtual bool data(void *) = 0;
    virtual vStrTab *strtab() const = 0;
    virtual Id index(uint n) const = 0;
    virtual vSymTab *transplant() = 0;
    inline Val get(Id id) const;
};
typedef const vSymTab &vSymTab_;

// external declaration requirements:
//  typedef unsigned int uint;
//  class vStrTab;
```

D.3 *CLex*

```
// this file was auto-generated by deco 0.12 from the command:
//  deco -fonlyinterface -fsavecode -include stdhdr.hh CLex-mi.hh

// types:
class CLex : public Lexer {
 public:
    enum Token {
        NONE = 0,
        EOF_COMPLETE = 1,
        EOF_INCOMPLETE = 2,
        INCOMPLETE = 3,
        INVALID = 4,
        ID = 257,
        NUM = 258,
        CHAR = 259,
        FLOAT = 260,
        STR = 261,
        CSTR = 262,
        BOOL = 263,
        BACKSTR = 264,
        ANGSTR = 265,
        LINE = 266,
        SPACE = 267,
        EOL = 268,
        CPP = 269,
        CPP_JOIN = 270,
        CPP_DEFINED = 271,
        CPP_DEFINE = 272,
        CPP_UNDEF = 273,
        CPP_IF = 274,
        CPP_IFDEF = 275,
        CPP_IFNDEF = 276,
        CPP_ENDIF = 277,
        CPP_ELIF = 278,
        CPP_INCLUDE = 279,
        CPP_PRAGMA = 280,
        CPP_ASSERT = 281,
        CPP_UNASSERT = 282,
        CPP_LINE = 283,
        CPP_ERROR = 284,
        CPP_WARNING = 285,
        CPP_IDENT = 286,
        CPP__FILE = 287,
        CPP__LINE = 288,
        CPP__DATE = 289,
        CPP__TIME = 290,
        CPP__STDC = 291,
        CPP__STDCVER = 292,
        EOS = 293,
```

```
COLON = 294,
COMMA = 295,
QMARK = 296,
POUND = 297,
LBRACE = 298,
RBRACE = 299,
LBRACK = 300,
RBRACK = 301,
LPAREN = 302,
RPAREN = 303,
QLINE = 304,
DOTDOT = 305,
AT = 306,
BSLASH = 307,
OP_ADD = 308,
OP_SUB = 309,
OP_MUL = 310,
OP_DIV = 311,
OP_MOD = 312,
OP_NOT = 313,
OP_AND = 314,
OP_OR = 315,
OP_XOR = 316,
OP_LNOT = 317,
OP_LAND = 318,
OP_LOR = 319,
OP_LXOR = 320,
OP_LSS = 321,
OP_GTR = 322,
OP_LEQ = 323,
OP_GEQ = 324,
OP_EQU = 325,
OP_NEQ = 326,
OP_INC = 327,
OP_DEC = 328,
OP_LSHIFT = 329,
OP_RSHIFT = 330,
OP_ASN = 331,
OP_ASNADD = 332,
OP_ASNSUB = 333,
OP_ASNMUL = 334,
OP_ASNDIV = 335,
OP_ASNMOD = 336,
OP_ASNAND = 337,
OP_ASNOR = 338,
OP_ASNXOR = 339,
OP_ASNLSHIFT = 340,
OP_ASNRSHIFT = 341,
OP_DOT = 342,
OP_PTRTO = 343,
OP_SCOPE = 344,
ELIPSIS = 345,
OP_MDOT = 346,
OP_MPTRTO = 347,
OP_CONDASN = 348,
KW_ATTRIBUTE = 349,
KW_CONST = 350,
KW_VOLATILE = 351,
KW_AUTO = 352,
KW_STATIC = 353,
KW_EXTERN = 354,
KW_REGISTER = 355,
KW_LONG = 356,
KW_SHORT = 357,
KW_SIGNED = 358,
KW_UNSIGNED = 359,
KW_INT = 360,
KW_CHAR = 361,
KW_FLOAT = 362,
KW_DOUBLE = 363,
KW_VOID = 364,
KW_IF = 365,
KW_ELSE = 366,
KW_DO = 367,
KW_WHILE = 368,
KW_FOR = 369,
KW_SWITCH = 370,
KW_CASE = 371,
KW_DEFAULT = 372,
```



```
KW_BREAK = 373,
KW_CONTINUE = 374,
KW_GOTO = 375,
KW_RETURN = 376,
KW_ENUM = 377,
KW_UNION = 378,
KW_STRUCT = 379,
KW_TYPEDEF = 380,
OP_SIZEOF = 381,
KW__FUNC = 382,
KW__PFUNC = 383,
KW_OPERATOR = 384,
KW_CLASS = 385,
KW_THIS = 386,
KW_TYPEID = 387,
KW_TYPENAME = 388,
KW_NAMESPACE = 389,
KW_USING = 390,
KW_TEMPLATE = 391,
KW_PUBLIC = 392,
KW_PRIVATE = 393,
KW_PROTECTED = 394,
KW_FRIEND = 395,
KW_VIRTUAL = 396,
KW_MUTABLE = 397,
KW_EXPLICIT = 398,
KW_BOOL = 399,
KW_TRUE = 400,
KW_FALSE = 401,
KW_WCHAR = 402,
KW_THROW = 403,
KW_CATCH = 404,
KW_TRY = 405,
KW_SCAST = 406,
KW_DCAST = 407,
KW_CCAST = 408,
KW_RCAST = 409,
KW_INLINE = 410,
KW_ASM = 411,
OP_TYPEOF = 412,
OP_NEW = 413,
OP_DELETE = 414,
OP_NEWA = 415,
OP_DELETEA = 416,
KW_DYNAMIC = 417,
VAL_ID = 418,
AGGR_ID = 419,
ENUM_ID = 420,
TYPE_ID = 421,
CTOR_ID = 422,
DTOR_ID = 423,
OPER_ID = 424,
CONV_ID = 425,
NSPACE_ID = 426,
ASPACE_ID = 427,
TPLATE_ID = 428,
SCOPE = 429,
TOKENLIST = 430,
OP_REF = 431,
OP_PTR = 432,
OP_MPTR = 433,
OP_UMINUS = 434,
OP_UPLUS = 435,
OP_INCP = 436,
OP_DECP = 437,
OP_TSIZEOF = 438,
OP_COMMA = 439,
OP_FUNC = 440,
OP_INDEX = 441,
OP_CAST = 442,
KW_DCLASS = 443,
KW_DTYPE = 444,
NEXT_INSN = 445,
BACKPATCH = 446,
ACTION = 447,
TBC = 448,
LBRACE_SAVE = 449,
COLON_SAVE = 450,
PARSE_COMPOUND = 451,
```

```
        DEFINE = 452,
        YYLASTTOKEN = 452,
        NUM_TOKENS = 453,
        START_CPP = 272,
        END_CPP = 292,
        START_C = 349,
        END_C = 383,
        START_CXX = 384,
        END_CXX = 414,
        START_DCXX = 417,
        END_DCXX = 417,
    };
    CLex(char *buf, size_t str_size);
    Token next_token();
    csz *token_string();
    csz *token_string(Token t) const;
    csz *token_string(Token t, bool tf) const;
    csz *token_string(Token t, uint64 val, bool is_signed = true, int is_long = 0);
    csz *token_string(Token t, csz *str, size_t len = ((size_t)-1));
    csz *token_spacing(Token prev, Token next) const;
    uint last_char() const;
    inline CLex &set_buf(char *ptr, size_t size);
    inline CLex &set_tok(char *ptr, size_t size);
    inline CLex &set_name(char *ptr, size_t size);
    inline CLex &parse_space(bool tf = true);
    inline CLex &parse_indent(bool tf = true);
    inline CLex &parse_keywords(bool tf = true);
    inline CLex &parse_c();
    inline CLex &parse_cpp(bool tf = true);
    inline CLex &parse_cxx();
    inline CLex &parse_dcxx();
    inline CLex &parse_eol(bool tf = true);
    inline void get_line();
    inline Token last_token() const;
    inline uint64 last_num() const;
    inline bool last_bool() const;
    inline bool last_signed() const;
    inline int last_long() const;
    inline double last_float() const;
    inline csz *last_str() const;
    inline csz *last_id() const;
    inline size_t last_size() const;
    inline csz *source() const;
    inline void source(csz *str);
    inline int include_level();
    inline bool in_string() const;
    inline bool in_char() const;
    inline bool in_bquote() const;
    inline void set_token(Token t);
};

// external declaration requirements:
//  typedef unsigned int uint;
//  typedef const char csz;

// external definition requirements:
//  typedef unsigned long long uint64;
//  typedef unsigned int size_t;
//  class Lexer;
```

D.4 *CParse*

```
// this file was auto-generated by deco 0.12 from the command:
//  deco -fonlyinterface -fsavecode -include stdhdr.hh CParse-mi.hh

// types:
class ActionStack;

class CParse {
 public:
```



```
    typedef vCAction::Item Item;
    typedef const vCAction::Item &Item_;
 protected:
    vCAction *_caction;
    bool _yydebug;
    Item _current_scope;
    Item _enum_type;
    Item _enum_val;
    Item _stmtlist;
    bool _savecode;
    bool _gsavecode;
    bool _rawid;
    TokenList *_deftok;
 public:
    CParse(CLex &clex, vCAction &caction);
    int parse();
    void inject(TokenList *tok);
    inline bool debug();
    inline void debug(bool d = true);
    inline bool yydebug();
    inline void yydebug(bool d = true);
    inline void savecode(bool b = true);
};

// external declaration requirements:
//  class CLex;
//  class vCAction;
//  class TokenList;
```

D.5 *vCAction*

```
// this file was auto-generated by deco 0.12 from the command:
//  deco -fonlyinterface -fsavecode -include stdhdr.hh vCAction-mi.hh

// types:
class vErrorReport {
 public:
    virtual void error(csz *fmt, ...) = 0;
    virtual void warning(csz *fmt, ...) = 0;
    virtual void abort(csz *fmt, ...) = 0;
    virtual void internal_error(csz *file, int line, csz *module = 0) = 0;
};
class TokenList;

class vCAction : public vErrorReport {
 public:
    enum Token {
        NONE = 0,
        EOF_COMPLETE = 1,
        EOF_INCOMPLETE = 2,
        INCOMPLETE = 3,
        INVALID = 4,
        ID = 257,
        NUM = 258,
        CHAR = 259,
        FLOAT = 260,
        STR = 261,
        CSTR = 262,
        BOOL = 263,
        BACKSTR = 264,
        ANGSTR = 265,
        LINE = 266,
        SPACE = 267,
        EOL = 268,
        CPP = 269,
        CPP_JOIN = 270,
        CPP_DEFINED = 271,
        CPP_DEFINE = 272,
        CPP_UNDEF = 273,
        CPP_IF = 274,
```

```
CPP_IFDEF = 275,
CPP_IFNDEF = 276,
CPP_ENDIF = 277,
CPP_ELIF = 278,
CPP_INCLUDE = 279,
CPP_PRAGMA = 280,
CPP_ASSERT = 281,
CPP_UNASSERT = 282,
CPP_LINE = 283,
CPP_ERROR = 284,
CPP_WARNING = 285,
CPP_IDENT = 286,
CPP__FILE = 287,
CPP__LINE = 288,
CPP__DATE = 289,
CPP__TIME = 290,
CPP__STDC = 291,
CPP__STDCVER = 292,
EOS = 293,
COLON = 294,
COMMA = 295,
QMARK = 296,
POUND = 297,
LBRACE = 298,
RBRACE = 299,
LBRACK = 300,
RBRACK = 301,
LPAREN = 302,
RPAREN = 303,
QLINE = 304,
DOTDOT = 305,
AT = 306,
BSLASH = 307,
OP_ADD = 308,
OP_SUB = 309,
OP_MUL = 310,
OP_DIV = 311,
OP_MOD = 312,
OP_NOT = 313,
OP_AND = 314,
OP_OR = 315,
OP_XOR = 316,
OP_LNOT = 317,
OP_LAND = 318,
OP_LOR = 319,
OP_LXOR = 320,
OP_LSS = 321,
OP_GTR = 322,
OP_LEQ = 323,
OP_GEQ = 324,
OP_EQU = 325,
OP_NEQ = 326,
OP_INC = 327,
OP_DEC = 328,
OP_LSHIFT = 329,
OP_RSHIFT = 330,
OP_ASN = 331,
OP_ASNADD = 332,
OP_ASNSUB = 333,
OP_ASNMUL = 334,
OP_ASNDIV = 335,
OP_ASNMOD = 336,
OP_ASNAND = 337,
OP_ASNOR = 338,
OP_ASNXOR = 339,
OP_ASNLSHIFT = 340,
OP_ASNRSHIFT = 341,
OP_DOT = 342,
OP_PTRTO = 343,
OP_SCOPE = 344,
ELIPSIS = 345,
OP_MDOT = 346,
OP_MPTRTO = 347,
OP_CONDASN = 348,
KW_ATTRIBUTE = 349,
KW_CONST = 350,
KW_VOLATILE = 351,
KW_AUTO = 352,
KW_STATIC = 353,
```



```
KW_EXTERN = 354,
KW_REGISTER = 355,
KW_LONG = 356,
KW_SHORT = 357,
KW_SIGNED = 358,
KW_UNSIGNED = 359,
KW_INT = 360,
KW_CHAR = 361,
KW_FLOAT = 362,
KW_DOUBLE = 363,
KW_VOID = 364,
KW_IF = 365,
KW_ELSE = 366,
KW_DO = 367,
KW_WHILE = 368,
KW_FOR = 369,
KW_SWITCH = 370,
KW_CASE = 371,
KW_DEFAULT = 372,
KW_BREAK = 373,
KW_CONTINUE = 374,
KW_GOTO = 375,
KW_RETURN = 376,
KW_ENUM = 377,
KW_UNION = 378,
KW_STRUCT = 379,
KW_TYPEDEF = 380,
OP_SIZEOF = 381,
KW__FUNC = 382,
KW__PFUNC = 383,
KW_OPERATOR = 384,
KW_CLASS = 385,
KW_THIS = 386,
KW_TYPEID = 387,
KW_TYPENAME = 388,
KW_NAMESPACE = 389,
KW_USING = 390,
KW_TEMPLATE = 391,
KW_PUBLIC = 392,
KW_PRIVATE = 393,
KW_PROTECTED = 394,
KW_FRIEND = 395,
KW_VIRTUAL = 396,
KW_MUTABLE = 397,
KW_EXPLICIT = 398,
KW_BOOL = 399,
KW_TRUE = 400,
KW_FALSE = 401,
KW_WCHAR = 402,
KW_THROW = 403,
KW_CATCH = 404,
KW_TRY = 405,
KW_SCAST = 406,
KW_DCAST = 407,
KW_CCAST = 408,
KW_RCAST = 409,
KW_INLINE = 410,
KW_ASM = 411,
OP_TYPEOF = 412,
OP_NEW = 413,
OP_DELETE = 414,
OP_NEWA = 415,
OP_DELETEA = 416,
KW_DYNAMIC = 417,
VAL_ID = 418,
AGGR_ID = 419,
ENUM_ID = 420,
TYPE_ID = 421,
CTOR_ID = 422,
DTOR_ID = 423,
OPER_ID = 424,
CONV_ID = 425,
NSPACE_ID = 426,
ASPACE_ID = 427,
TPLATE_ID = 428,
SCOPE = 429,
TOKENLIST = 430,
OP_REF = 431,
OP_PTR = 432,
```

```
        OP_MPTR = 433,
        OP_UMINUS = 434,
        OP_UPLUS = 435,
        OP_INCP = 436,
        OP_DECP = 437,
        OP_TSIZEOF = 438,
        OP_COMMA = 439,
        OP_FUNC = 440,
        OP_INDEX = 441,
        OP_CAST = 442,
        KW_DCLASS = 443,
        KW_DTYPE = 444,
        NEXT_INSN = 445,
        BACKPATCH = 446,
        ACTION = 447,
        TBC = 448,
        LBRACE_SAVE = 449,
        COLON_SAVE = 450,
        PARSE_COMPOUND = 451,
        DEFINE = 452,
        YYLASTTOKEN = 452,
        TEMP = 453,
        LITERAL = 454,
        RESULT = 455,
        INSN = 456,
        LIST = 457,
        TREE = 458,
        ARRAY = 459,
        OBJ = 460,
        TYPE = 461,
        TYPELIST = 462,
        PARM = 463,
        PARMLIST = 464,
        EXPR = 465,
        EXPRLIST = 466,
        STMT = 467,
        STMTLIST = 468,
        ERR = 469,
        ERRLIST = 470,
        DECL = 471,
        DECLLIST = 472,
        STRUCT = 473,
        CLASS = 474,
        VAR = 475,
        FUNC = 476,
        CTORINIT = 477,
        NUM_TOKENS = 478,
    };
    struct Item {
        union {
            struct {
                bool is_signed;
                char is_long;
            } c;
            vSymTab *scope;
            void *cptr;
        };
        union {
            struct {
                int32 top;
                int32 bottom;
            } Action;
            uint64 val;
            int64 sval;
            Token tval;
            uint64 Num;
            bool Bool;
            uint Char;
            vStrTab::Id id;
            double Float;
            class TokenList *TokenList;
        };
        Token token;
        ushort line;
        inline Item();
        inline Item(Token t);
        inline Item(CLex::Token t);
        inline Item(Token t, uint64 v);
        inline Item(CLex::Token t, uint64 v);
```



```
        inline Item(int n);
        inline Item &operator=(Token t);
        inline Item &operator=(CLex::Token t);
    };
    typedef const Item &Item_;
    typedef void action(Item *r, Item_ p1, Item_ p2, Item_ p3);
    typedef action (::vCAction ::*Action);
protected:
    bool _complete;
    int _num_errors;
    int _num_warnings;
public:
    static const Item none;
    inline bool complete() const;
    inline void complete(bool tf);
    inline int errors() const;
    inline int warnings() const;
    virtual void accept() = 0;
    virtual void get_input(bool split_token) = 0;
    virtual vStrTab::Id strid(Token, csz *str) = 0;
    virtual void eol() = 0;
    virtual csz *cstr(Token token) const = 0;
    virtual csz *cstr(vStrTab::Id id) const = 0;
    virtual sz *cstr(Item_ item, sz *buf, size_t buflen, bool show_type = false) const = 0;
    virtual void eprintf(csz *fmt, ...) = 0;
    virtual void veprintf(csz *fmt, va_list args) = 0;
    virtual void mprintf(csz *prefix, csz *fmt, ...) = 0;
    virtual void vmprintf(csz *prefix, csz *fmt, va_list args) = 0;
    virtual void _assertfail(csz *cond, csz *filename, int line) = 0;
    virtual void mprint_action(csz *prefix, csz *name, Item_ p1, Item_ p2, Item_ p3) = 0;
    virtual void mprint_result(csz *prefix, Item_ r) = 0;
    virtual void unsupported(csz *name, Item_ p1, Item_ p2, Item_ p3) = 0;
    virtual vSymTab *get_symtab(Item_ id, Item_ scope) = 0;
    virtual void Test(Item *r, Item_ p1, Item_ p2, Item_ p3) = 0;
    virtual void Undeclared(Item *r, Item_ sym, Item_ scope, Item_) = 0;
    virtual void Literal(Item *r, Item_ item, Item_, Item_) = 0;
    virtual void Primary(Item *r, Item_ item, Item_ scope, Item_) = 0;
    virtual void UnaryOp(Item *r, Item_ op, Item_ arg, Item_) = 0;
    virtual void BinaryOp(Item *r, Item_ op, Item_ arg1, Item_ arg2) = 0;
    virtual void Index(Item *r, Item_ array, Item_ index, Item_) = 0;
    virtual void DeclParam(Item *r, Item_ type, Item_ sym, Item_) = 0;
    virtual void DeclSym(Item *r, Item_ type, Item_ sym, Item_ scope) = 0;
    virtual void DeclPtr(Item *r, Item_ ptrtype, Item_ type, Item_) = 0;
    virtual void DeclArray(Item *r, Item_ type, Item_ size, Item_ lower) = 0;
    virtual void DeclFunc(Item *r, Item_ type, Item_ plist, Item_ cv) = 0;
    virtual void SetAttr(Item *r, Item_ decl, Item_ attr, Item_ val) = 0;
    virtual void Define(Item *r, Item_ var, Item_ val, Item_) = 0;
    virtual void DefParam(Item *r, Item_ par, Item_ val, Item_) = 0;
    virtual void AggrRef(Item *r, Item_ type, Item_ sym, Item_ scope) = 0;
    virtual void AggrDecl(Item *r, Item_ type, Item_ sym, Item_ scope) = 0;
    virtual void NewAggr(Item *r, Item_ type, Item_ sym, Item_ scope) = 0;
    virtual void EndAggr(Item *r, Item_ aggr, Item_, Item_) = 0;
    virtual void NewNspace(Item *r, Item_ type, Item_ sym, Item_ scope) = 0;
    virtual void NewLocal(Item *r, Item_ type, Item_ sym, Item_ scope) = 0;
    virtual void FuncScope(Item *r, Item_ sym, Item_, Item_) = 0;
    virtual void NewScope(Item *r, Item_ type, Item_ sym, Item_ scope) = 0;
    virtual void FreeScope(Item *r, Item_ scope, Item_, Item_) = 0;
    virtual void EnumRef(Item *r, Item_ type, Item_ sym, Item_ scope) = 0;
    virtual void NewEnum(Item *r, Item_ type, Item_ sym, Item_ scope) = 0;
    virtual void EnumNext(Item *r, Item_ prev, Item_, Item_) = 0;
    virtual void SetAccess(Item *r, Item_ scope, Item_ acc, Item_) = 0;
    virtual void SetLinkage(Item *r, Item_ scope, Item_ str, Item_) = 0;
    virtual void Inherit(Item *r, Item_ scope, Item_ itype, Item_ base) = 0;
    virtual void BuildTypeList(Item *r, Item_ typelist, Item_ item, Item_) = 0;
    virtual void BuildParamList(Item *r, Item_ plist, Item_ item, Item_) = 0;
    virtual void BuildArray(Item *r, Item_ exprlist, Item_ item, Item_) = 0;
    virtual void BuildExprList(Item *r, Item_ exprlist, Item_ item, Item_) = 0;
    virtual void BuildStmtList(Item *r, Item_ stmtlist, Item_ item, Item_) = 0;
    virtual void FindSym(Item *r, Item_ type, Item_ sym, Item_ scope) = 0;
    virtual void Cast(Item *r, Item_ type, Item_ val, Item_) = 0;
    virtual void Call(Item *r, Item_ fn, Item_ param, Item_ scope) = 0;
    virtual void SpecialId(Item *r, Item_ type, Item_ val, Item_ scope) = 0;
    virtual void SpecialStr(Item *r, Item_ type, Item_ scope, Item_) = 0;
    virtual void CtorInit(Item *r, Item_ id, Item_ params, Item_) = 0;
    virtual void CtorCall(Item *r, Item_ fn, Item_ param, Item_) = 0;
    virtual void OpNew(Item *r, Item_ type, Item_ init, Item_ param) = 0;
    virtual void Label(Item *r, Item_ a, Item_ b, Item_ c) = 0;
    virtual void Switch(Item *r, Item_ a, Item_ b, Item_ c) = 0;
    virtual void Return(Item *r, Item_ a, Item_ b, Item_ c) = 0;
```

```
    virtual void Goto(Item *r, Item_ a, Item_ b, Item_ c) = 0;
    virtual void Break(Item *r, Item_ a, Item_ b, Item_ c) = 0;
    virtual void Continue(Item *r, Item_ a, Item_ b, Item_ c) = 0;
    virtual void IfElse(Item *r, Item_ a, Item_ b, Item_ c) = 0;
    virtual void CondExpr(Item *r, Item_ a, Item_ b, Item_ c) = 0;
    virtual void While(Item *r, Item_ a, Item_ b, Item_ c) = 0;
    virtual void DoWhile(Item *r, Item_ a, Item_ b, Item_ c) = 0;
    virtual void For(Item *r, Item_ a, Item_ b, Item_ c) = 0;
    virtual void ForExpr(Item *r, Item_ a, Item_ b, Item_ c) = 0;
};

// external declaration requirements:
//  typedef char sz;
//  typedef const char csz;
//  typedef unsigned int size_t;
//  typedef __gnuc_va_list va_list;
//  class vSymTab;

// external definition requirements:
//  typedef long int32;
//  typedef long long int64;
//  typedef unsigned long long uint64;
//  typedef unsigned short ushort;
//  typedef unsigned int uint;
```

D.6 *CObj*

```
// this file was auto-generated by deco 0.12 from the command:
//  deco -fonlyinterface -fsavecode -include stdhdr.hh CObj-mi.hh

// types:
typedef size_t bitsize_t;
class Object;
typedef Object obj_t;
typedef Object *type_t;

class Integral {
 public:
    Integral(int bits, int alignment, bool is_signed, bool is_bigend);
    inline bool is_signed() const;
    inline bool is_unsigned() const;
    inline size_t bitsize() const;
    inline size_t size() const;
    inline uint alignment() const;
    inline bool is_big_endian() const;
    inline bool is_little_endian() const;
    static inline type_t type();
};

class PtrTo {
 public:
    inline PtrTo(type_t type, bool c = false, bool v = false);
    inline type_t rtype() const;
    inline bool is_const();
    inline bool is_volatile();
    static inline type_t type();
};

class RefTo {
 public:
    inline RefTo(type_t type, bool c = false, bool v = false);
    inline type_t rtype() const;
    inline bool is_const();
    inline bool is_volatile();
    static inline type_t type();
};

class Qualifier {
 public:
    inline Qualifier(type_t type, bool c = false, bool v = false);
```



```
    inline type_t rtype() const;
    inline bool is_const();
    inline bool is_volatile();
    static inline type_t type();
};

class Array {
 public:
    inline Array(type_t type, size_t size, void *dsize = 0);
    inline size_t size() const;
    inline void *dsize() const;
    inline size_t lower_bound() const;
    inline size_t upper_bound() const;
    inline type_t rtype() const;
    static inline type_t type();
};

class Params {
 public:
    inline Params(int nparam, const type_t *ptype, bool variable = false);
    inline bool is_variable() const;
    inline uint nparam() const;
    inline type_t ptype(uint index) const;
    static inline type_t type();
};

class Func {
 public:
    inline Func(type_t rtype, type_t param, vSymTab *scope);
    inline bool is_variable() const;
    inline uint nparam() const;
    inline type_t ptype(uint n) const;
    inline type_t rtype() const;
    inline type_t psig() const;
    inline vSymTab *scope() const;
    static inline type_t type();
    inline void change_rtype(type_t type);
    inline void change_scope(vSymTab *scope);
};

class Method {
 public:
    inline Method(type_t otype, type_t ftype);
    inline type_t ftype() const;
    inline type_t otype() const;
    static inline type_t type();
};

struct ObjAttr {
    vStrTab::Id name;
    type_t type;
    bitsize_t offset;
    bitsize_t size;
};

struct AggrAttr {
    vStrTab::Id name;
    obj_t val;
};

class Aggr {
 public:
    inline Aggr(vStrTab::Id name, vSymTab *scope);
    inline vStrTab::Id name() const;
    inline vSymTab *scope() const;
    inline void set_scope(vSymTab *scope);
    static inline type_t type();
};

class Object {
 public:
    enum StorageType {
        Inline = 0,
        Auto = 1,
        Alloc = 2,
        _Reserved = 3,
    };
 protected:
    union {
```

```
        struct __anon_struct_1__ {
            void *ptr;
            uint32 count;
        };
        uint64 _val;
        double _fval;
        __anon_struct_1__ _ref;
    };
    vaddr_t _type;
    obj_t *_refcnt;
 public:
    bool is_of() const;
    size_t _sizeof() const;
    obj_t operator()(obj_t obj);
    obj_t operator()(obj_t *obj);
 protected:
    inline Object(type_t type, StorageType st);
    inline Object(obj_t &type, StorageType st);
 public:
    inline Object();
    inline type_t type() const;
    inline StorageType storagetype() const;
    inline bool is_null() const;
    inline bool is_type(obj_t &t) const;
    inline bool is_type(type_t t) const;
    inline bool is_inline() const;
    inline bool is_auto() const;
    inline bool is_alloc() const;
    inline void *ptr();
    inline const void *ptr() const;
    inline uint64 data() const;
    inline operator bool();
};

class obj_Integral : public Object {
 public:
    inline obj_Integral(int bits, int alignment, bool is_signed, bool is_bigend);
    inline operator Integral();
    inline obj_Integral(const Integral &data);
    inline obj_Integral(Integral *data);
};

class obj_PtrTo : public Object {
 public:
    size_t _sizeof() const;
    inline obj_PtrTo(type_t type, bool c = false, bool v = false);
    inline obj_PtrTo(const PtrTo &data);
    inline obj_PtrTo(PtrTo *data);
};

class obj_RefTo : public Object {
 public:
    inline obj_RefTo(type_t type, bool c = false, bool v = false);
    inline obj_RefTo(const RefTo &data);
    inline obj_RefTo(RefTo *data);
};

class obj_Qualifier : public Object {
 public:
    size_t _sizeof() const;
    inline obj_Qualifier(type_t type, bool c = false, bool v = false);
    inline obj_Qualifier(const Qualifier &data);
    inline obj_Qualifier(Qualifier *data);
};

class obj_Array : public Object {
 public:
    inline obj_Array(type_t type, size_t size, void *dsize = 0);
    inline obj_Array(const Array &data);
    inline obj_Array(Array *data);
};

class obj_Params : public Object {
 public:
    inline obj_Params(int nparam, const type_t *ptype, bool variable = false);
    inline obj_Params(const Params &data);
    inline obj_Params(Params *data);
};
```



```
class obj_Func : public Object {
 public:
    inline obj_Func(type_t rtype, type_t param, vSymTab *scope);
    inline obj_Func(const Func &data);
    inline obj_Func(Func *data);
};

class obj_Method : public Object {
 public:
    size_t _sizeof() const;
    inline obj_Method(type_t otype, type_t ftype);
    inline obj_Method(const Method &data);
    inline obj_Method(Method *data);
};

class obj_Aggr : public Object {
 public:
    size_t _sizeof() const;
    inline obj_Aggr(vStrTab::Id name, vSymTab *scope);
    inline obj_Aggr(const Aggr &data);
    inline obj_Aggr(Aggr *data);
};

// external:
extern obj_t Type_Integral;
extern obj_t Type_PtrTo;
extern obj_t Type_RefTo;
extern obj_t Type_Qualifier;
extern obj_t Type_Array;
extern obj_t Type_Params;
extern obj_t Type_Func;
extern obj_t Type_Method;
extern obj_t Type_Aggr;
extern obj_Integral Type_char;
extern obj_Integral Type_int;
extern obj_Integral Type_short;
extern obj_Integral Type_long;
extern obj_Integral Type_long2;
extern obj_Integral Type_schar;
extern obj_Integral Type_uchar;
extern obj_Integral Type_uint;
extern obj_Integral Type_ushort;
extern obj_Integral Type_ulong;
extern obj_Integral Type_ulong2;
extern obj_t Type_void;
extern obj_t Type_bool;
extern obj_t Type_ptr;
extern obj_t Type_ref;
extern obj_t Type_aggr;
extern obj_t Type_List;
extern obj_t Type_TokenList;
extern obj_t Type_Str;
extern obj_t Type_wchar;
extern obj_t Type_float;
extern obj_t Type_double;
extern obj_t Type_null;

// local exports/inline:
inline void *operator new(size_t, void *p);
inline Integral *dcast_Integral(Object *o);
inline PtrTo *dcast_PtrTo(Object *o);
inline RefTo *dcast_RefTo(Object *o);
inline Qualifier *dcast_Qualifier(Object *o);
inline Array *dcast_Array(Object *o);
inline Params *dcast_Params(Object *o);
inline Func *dcast_Func(Object *o);
inline Method *dcast_Method(Object *o);
inline Aggr *dcast_Aggr(Object *o);

// external declaration requirements:
//  typedef unsigned int uint;
//  class vSymTab;

// external definition requirements:
//  typedef unsigned char uint8;
//  typedef unsigned short uint16;
//  typedef unsigned long uint32;
//  typedef unsigned long long uint64;
//  typedef uint32 vaddr_t;
```

```
//  typedef unsigned int size_t;
```

D.7 *COut*

```
// this file was auto-generated by deco 0.12 from the command:
//  deco -fonlyinterface -fsavecode -include stdhdr.hh COut-mi.hh

// types:
class CCOut {
 public:
    enum DeclTypeMask {
        DTM_FN = 1,
        DTM_TYPE = 2,
        DTM_VAR = 4,
        DTM_CONST = 8,
        DTM_EXP = 16,
        DTM_IMP = 32,
        DTM_LOC = 64,
    };
    CCOut(vErrorReport &err, const Scope, DeclCommon::Access access = ((DeclCommon::Access)0));
    void printexpr(Expr, bool paren = false);
    void printstmt(Stmt);
    void printstmti(Stmt);
    void printexprlist(ExprList);
    void printstmtlist(StmtList);
    void printdecllist(StmtList);
    void printctorinit(StmtList);
    sz *literal_string(type_t type, uint64 val, sz *s);
    sz *type_string(type_t type, sz *s);
    sz *dbg_type_string(type_t type, sz *s);
    bool print_decls(const Scope sc, bool checkdep, bool err, int dtmask);
    void print_decls(const Scope, DeclCommon::Access acc = ((DeclCommon::Access)0));
    void print_defs(const Scope);
    void print_reqs(const Scope);
    sz *scoped_string(const Scope, sz *s);
    sz *scoped_string(Decl *decl, sz *s, bool needtag = false);
    sz *fullscoped_string(Decl *decl, sz *s);
    void onl(int y = 1);
    void onltab(int x = -1);
    int oprintf(csz *fmt, ...);
    inline void debugtypes(bool d = true);
    inline bool print_types(const Scope sc);
    inline void print_fn_imports(const Scope sc);
    inline void print_fn_exports(const Scope sc);
    inline void print_fn_local(const Scope sc);
    inline bool print_constants(const Scope sc);
    inline void print_var_imports(const Scope sc);
    inline void print_var_exports(const Scope sc);
    inline void fullscope(bool tf = true);
    inline void rshift();
    inline void lshift();
    inline void sectionlabel(csz *label);
    inline void width(int w);
    inline int width() const;
    inline void outmask(int m);
    inline int outmask() const;
    inline Scope scope() const;
    inline void scope(Scope s);
    inline void interface(bool tf);
    inline void onlyinterface(bool tf);
    inline bool interface() const;
    inline bool onlyinterface() const;
    inline void hdr(bool tf);
    inline bool hdr() const;
};

// external:
sz *strpcat(sz *str, csz *prefix);
csz *simple_typename(type_t type);
```



```
// external declaration requirements:
//  typedef unsigned long long uint64;
//  typedef char sz;
//  typedef const char csz;
//  class vStrTab;
//  typedef Object *type_t;
//  class vErrorReport;
//  class TokenList;
//  struct Decl;
//  class Expr;
//  class ExprList;
//  class Stmt;
//  class StmtList;

// external definition requirements:
//  class CLex;
//  class Scope;
```

E

Performance Tests

E.1 Source Code

E.1.1 *perf.dc*

This is the original DC++ source code that was used for the performance tests.

```
// td1 - types and classes

//#include <dcobj>
//#include <stdlib.h>

// defined here since DECO has some problems parsing the
// latest Linux/glibc system headers:
struct timeval {
    long tv_sec;
    long tv_usec;
};
extern "C" int gettimeofday(struct timeval *tv, void *tz);
extern "C" int printf(const char *fmt, ...);


type dt1 {
    int f();
    int g(int n);
};


class dc1 {
    int *v;
public:
    dc1(int *p) { v = p; }
dt1:
    int f() { return 3; }
    int g(int n) { return n * *v; }
};


class v1 {
public:
    virtual int f() = 0;
    virtual int g(int n) = 0;
};


class cv1 : public v1 {
    int *v;
```



```
public:
    cv1(int *p) { v = p; }
    int f() { return 3; }
    int g(int n) { return n * *v; }
};


int
test1(int n, dt1 o)
{
    int t = n*3;
    while(n--)
        t -= o.f();
    if(t != 0)
        printf("ERROR\n");
}

int
test2(int n, v1 *o)
{
    int t = n*3;
    while(n--)
        t -= o->f();
    if(t != 0)
        printf("ERROR\n");
}

int
test3(int n, dt1 o)
{
    int t = n*9*7;
    while(n--)
        t -= o.g(7);
    if(t != 0)
        printf("ERROR\n");
}

int
test4(int n, v1 *o)
{
    int t = n*9*7;
    while(n--)
        t -= o->g(7);
    if(t != 0)
        printf("ERROR\n");
}

long
timediff(struct timeval *tv0)
{
    timeval tvnow;
    gettimeofday(&tvnow, 0);
    return (tvnow.tv_sec-tv0->tv_sec)*1000000+(tvnow.tv_usec-tv0->tv_usec);
}

int
main()
{
    int x = 9;
    dc1 dc(&x);
    cv1 vc(&x);

    for(int n=1; n<100000; n += ((n+3)/4)) {
        int c = 1000000/n;
        timeval t0;
        printf("%d\t", n);
        gettimeofday(&t0, 0);
        for(int i=0; i<c; i++)
            test1(n, dc);
        printf("%d\t", timediff(&t0));
        gettimeofday(&t0, 0);
        for(int i=0; i<c; i++)
            test2(n, &vc);
        printf("%d\t", timediff(&t0));
        gettimeofday(&t0, 0);
        for(int i=0; i<c; i++)
            test3(n, dc);
        printf("%d\t", timediff(&t0));
        gettimeofday(&t0, 0);
```

```
        for(int i=0; i<c; i++)
            test4(n, &vc);
        printf("%d\n", timediff(&t0));
    }
}
```


E.1.2 *perf.cc*

This is the result of running *perf.dc* through DECO.

```
// this file was auto-generated by deco 0.13 from the command:
//  deco perf.dc

// types:
class __dchdr__;

class BTRef {
 public:
    void **_itabp;
    void *self;
    void *_iobj;
    void *_cobj;
};

struct Uuid {
    char *_str;
    inline bool operator==(Uuid *u);
    inline bool operator!=(Uuid *u);
};

struct timeval {
    long tv_sec;
    long tv_usec;
};

class v1 {
 public:
    virtual int f() = 0;
    virtual int g(int n) = 0;
};

class cv1 : public v1 {
    int *v;
 public:
    inline cv1(int *p);
    inline int f();
    inline int g(int n);
};

class any : public BTRef {
 public:
    bool default__dcbind(BTRef *_a0_, Uuid tid);
    void default__dbound();
    void default__dunbound();
    void default__dunbind();
    inline bool dcbind(BTRef *_a0_, Uuid tid);
    inline void dbound();
    inline void dunbound();
    inline void dunbind();
};

class dt1 : public BTRef {
 public:
    int default__f();
    int default__g(int n);
    operator any() const;
    inline int f();
    inline int g(int n);
    inline bool dcbind(BTRef *_a0_, Uuid tid);
    inline void dbound();
    inline void dunbound();
    inline void dunbind();
};

class dc1 {
 public:
    class dt1 : public ::dt1 {
        friend dc1;
        inline dt1() {}
     public:
        inline dc1 *iobj();
        inline int f();
        inline int g(int n);
```

```
    };
    class _type : public BTRef {
     public:
        operator dt1() const;
        operator any() const;
        inline int f();
        inline int g(int n);
        inline bool dcbind(BTRef *_a0_, Uuid tid);
        inline void dbound();
        inline void dunbound();
        inline void dunbind();
    };
 private:
    int *v;
    static void *_itable[];
 public:
    inline dc1(int *p);
    inline int dt1__f();
    inline int dt1__g(int n);
    void dbind(any *btrp) const;
    inline operator any() const;
    inline bool default__dcbind(BTRef *_a0_, Uuid tid);
    inline void default__dbound();
    inline void default__dunbound();
    inline void default__dunbind();
    void dbind(::dt1 *btrp) const;
    inline operator ::dt1() const;
};

// imports:
extern "C" int gettimeofday(timeval *tv, void *tz);
extern "C" int printf(const char *fmt, ...);

// exports:
int test1(int n, /*type*/ dt1 o);
int test2(int n, v1 *o);
int test3(int n, /*type*/ dt1 o);
int test4(int n, v1 *o);
long timediff(timeval *tv0);
int main();

/*inline*/ bool
Uuid::operator==(Uuid *u)
{
    return this->_str == u->_str;
}

/*inline*/ bool
Uuid::operator!=(Uuid *u)
{
    return !(*this == u);
}

/*inline*/
cv1::cv1(int *p)
{
    this->v = p;
}

/*inline*/ int
cv1::f()
{
    return 3;
}

/*inline*/ int
cv1::g(int n)
{
    return n**this->v;
}

/*inline*/ bool
any::dcbind(BTRef *_a0_, Uuid tid)
{
    return ((bool (*)(any *, BTRef *, Uuid))(BTRef::_itabp[0]))(this, _a0_, tid);
}

/*inline*/ void
any::dbound()
```



```
{
    ((void (*)(any *))(BTRef::_itabp[1]))(this);
}

/*inline*/ void
any::dunbound()
{
    ((void (*)(any *))(BTRef::_itabp[2]))(this);
}

/*inline*/ void
any::dunbind()
{
    ((void (*)(any *))(BTRef::_itabp[3]))(this);
}

/*inline*/ int
dt1::f()
{
    return ((int (*)(dt1 *))(BTRef::_itabp[0]))(this);
}

/*inline*/ int
dt1::g(int n)
{
    return ((int (*)(dt1 *, int))(BTRef::_itabp[1]))(this, n);
}

/*inline*/ bool
dt1::dcbind(BTRef *_a0_, Uuid tid)
{
    return ((bool (*)(dt1 *, BTRef *, Uuid))(reinterpret_cast<void **>(BTRef::_itabp[2])[0]))(this, _a0_, tid);
}

/*inline*/ void
dt1::dbound()
{
    ((void (*)(dt1 *))(reinterpret_cast<void **>(BTRef::_itabp[2])[1]))(this);
}

/*inline*/ void
dt1::dunbound()
{
    ((void (*)(dt1 *))(reinterpret_cast<void **>(BTRef::_itabp[2])[2]))(this);
}

/*inline*/ void
dt1::dunbind()
{
    ((void (*)(dt1 *))(reinterpret_cast<void **>(BTRef::_itabp[2])[3]))(this);
}

void *dc1::_itable[] = {
    (void *)(&::any::dcbind),
    (void *)(&::any::dbound),
    (void *)(&::any::dunbound),
    (void *)(&::any::dunbind),
    (void *)(&dt1::f),
    (void *)(&dt1::g),
    reinterpret_cast<void *>(_itable)
};

/*inline*/ dc1 *
dc1::dt1::iobj()
{
    return reinterpret_cast<dc1 *>(_iobj);
}

/*inline*/ int
dc1::dt1::f()
{
    return iobj()->dt1__f();
}

/*inline*/ int
dc1::dt1::g(int n)
{
    return iobj()->dt1__g(n);
}
```

```
/*inline*/ int
dc1::_type::f()
{
    return ((int (*)(::dc1::_type *))(reinterpret_cast<void **>(BTRef::_itabp[0])[0]))(this);
}

/*inline*/ int
dc1::_type::g(int n)
{
    return ((int (*)(::dc1::_type *, int))(reinterpret_cast<void **>(BTRef::_itabp[0])[1]))(this, n);
}

/*inline*/ bool
dc1::_type::dcbind(BTRef *_a0_, Uuid tid)
{
    return ((bool (*)(::dc1::_type *, BTRef *, Uuid))(reinterpret_cast<void **>(BTRef::_itabp[1])[0]))(th
}

/*inline*/ void
dc1::_type::dbound()
{
    ((void (*)(::dc1::_type *))(reinterpret_cast<void **>(BTRef::_itabp[1])[1]))(this);
}

/*inline*/ void
dc1::_type::dunbound()
{
    ((void (*)(::dc1::_type *))(reinterpret_cast<void **>(BTRef::_itabp[1])[2]))(this);
}

/*inline*/ void
dc1::_type::dunbind()
{
    ((void (*)(::dc1::_type *))(reinterpret_cast<void **>(BTRef::_itabp[1])[3]))(this);
}

/*inline*/
dc1::dc1(int *p)
{
    this->v = p;
}

/*inline*/ int
dc1::dt1__f()
{
    return 3;
}

/*inline*/ int
dc1::dt1__g(int n)
{
    return n**this->v;
}

void
dc1::dbind(any *btrp) const
{
    btrp->_iobj = reinterpret_cast<void *>(this);
    btrp->_itabp = _itable;
}

/*inline*/
dc1::operator any() const
{
    any btr;
    dbind(&btr);
    return btr;
}

/*inline*/ bool
dc1::default__dcbind(BTRef *_a0_, Uuid tid)
{
    return static_cast< ::any>(*this).default__dcbind(_a0_, tid);
}

/*inline*/ void
dc1::default__dbound()
{
    static_cast< ::any>(*this).default__dbound();
```



```
}

/*inline*/ void
dc1::default__dunbound()
{
    static_cast< ::any>(*this).default__dunbound();
}

/*inline*/ void
dc1::default__dunbind()
{
    static_cast< ::any>(*this).default__dunbind();
}

void
dc1::dbind(::dt1 *btrp) const
{
    btrp->_iobj = reinterpret_cast<void *>(this);
    btrp->_itabp = &_itable[4];
}

/*inline*/
dc1::operator ::dt1() const
{
    ::dt1 btr;
    dbind(&btr);
    return btr;
}

int
test1(int n, /*type*/ dt1 o)
{
    int t = n*3;
    while(n--)
        t -= o.f();
    if(t != 0)
        printf("ERROR\n");
}

int
test2(int n, v1 *o)
{
    int t = n*3;
    while(n--)
        t -= o->f();
    if(t != 0)
        printf("ERROR\n");
}

int
test3(int n, /*type*/ dt1 o)
{
    int t = n*9*7;
    while(n--)
        t -= o.g(7);
    if(t != 0)
        printf("ERROR\n");
}

int
test4(int n, v1 *o)
{
    int t = n*9*7;
    while(n--)
        t -= o->g(7);
    if(t != 0)
        printf("ERROR\n");
}

long
timediff(timeval *tv0)
{
    timeval tvnow;
    gettimeofday(&tvnow, 0);
    return (tvnow.tv_sec-tv0->tv_sec)*1000000+(tvnow.tv_usec-tv0->tv_usec);
}

int
main()
```

```
{
    int x = 9;
    dc1 dc(&x);
    cv1 vc(&x);
    for(int n = 1; n < 100000; n += (n+3)/4) {
        int c = 1000000/n;
        timeval t0;
        printf("%d\t", n);
        gettimeofday(&t0, 0);
        for(int i = 0; i < c; i++)
            test1(n, dc);
        printf("%d\t", timediff(&t0));
        gettimeofday(&t0, 0);
        for(int i = 0; i < c; i++)
            test2(n, &vc);
        printf("%d\t", timediff(&t0));
        gettimeofday(&t0, 0);
        for(int i = 0; i < c; i++)
            test3(n, dc);
        printf("%d\t", timediff(&t0));
        gettimeofday(&t0, 0);
        for(int i = 0; i < c; i++)
            test4(n, &vc);
        printf("%d\n", timediff(&t0));
    }
}
```


E.1.3 *perf2.cc*, (*perf.cc with manual inlining fix)*

This is a copy of *perf.cc* that was manually modified to fix the order of the `f()` and `g()` functions with respect to the stubs that call them, so that inlining would properly occur.

```
// this file was auto-generated by deco 0.13 from the command:
//  ./deco perf.dc

// types:
class __dchdr__;

class BTRef {
 public:
    void **_itabp;
    void *self;
    void *_iobj;
    void *_cobj;
};

struct Uuid {
    char *_str;
    inline bool operator==(Uuid *u);
    inline bool operator!=(Uuid *u);
};

struct timeval {
    long tv_sec;
    long tv_usec;
};

class v1 {
 public:
    virtual int f() = 0;
    virtual int g(int n) = 0;
};

class cv1 : public v1 {
    int *v;
 public:
    inline cv1(int *p);
    inline int f();
    inline int g(int n);
};

class any : public BTRef {
 public:
    bool default__dcbind(BTRef *_a0_, Uuid tid);
    void default__dbound();
    void default__dunbound();
    void default__dunbind();
    inline bool dcbind(BTRef *_a0_, Uuid tid);
    inline void dbound();
    inline void dunbound();
    inline void dunbind();
};

class dt1 : public BTRef {
 public:
    int default__f();
    int default__g(int n);
    operator any() const;
    inline int f();
    inline int g(int n);
    inline bool dcbind(BTRef *_a0_, Uuid tid);
    inline void dbound();
    inline void dunbound();
    inline void dunbind();
};

class dc1 {
 public:
    class dt1 : public ::dt1 {
        friend dc1;
        inline dt1() {}
     public:
        inline dc1 *iobj();
```

```
        inline int f();
        inline int g(int n);
    };
    class _type : public BTRef {
     public:
        operator dt1() const;
        operator any() const;
        inline int f();
        inline int g(int n);
        inline bool dcbind(BTRef *_a0_, Uuid tid);
        inline void dbound();
        inline void dunbound();
        inline void dunbind();
    };
 private:
    int *v;
    static void *_itable[];
 public:
    inline dc1(int *p);
    inline int dt1__f();
    inline int dt1__g(int n);
    void dbind(any *btrp) const;
    inline operator any() const;
    inline bool default__dcbind(BTRef *_a0_, Uuid tid);
    inline void default__dbound();
    inline void default__dunbound();
    inline void default__dunbind();
    void dbind(::dt1 *btrp) const;
    inline operator ::dt1() const;
};

// imports:
extern "C" int gettimeofday(timeval *tv, void *tz);
extern "C" int printf(const char *fmt, ...);

// exports:
int test1(int n, /*type*/ dt1 o);
int test2(int n, v1 *o);
int test3(int n, /*type*/ dt1 o);
int test4(int n, v1 *o);
long timediff(timeval *tv0);
int main();

/*inline*/ bool
Uuid::operator==(Uuid *u)
{
    return this->_str == u->_str;
}

/*inline*/ bool
Uuid::operator!=(Uuid *u)
{
    return !(*this == u);
}

/*inline*/
cv1::cv1(int *p)
{
    this->v = p;
}

/*inline*/ int
cv1::f()
{
    return 3;
}

/*inline*/ int
cv1::g(int n)
{
    return n**this->v;
}

/*inline*/ bool
any::dcbind(BTRef *_a0_, Uuid tid)
{
    return ((bool (*)(any *, BTRef *, Uuid))(BTRef::_itabp[0]))(this, _a0_, tid);
}
```



```
/*inline*/ void
any::dbound()
{
    ((void (*)(any *))(BTRef::_itabp[1]))(this);
}

/*inline*/ void
any::dunbound()
{
    ((void (*)(any *))(BTRef::_itabp[2]))(this);
}

/*inline*/ void
any::dunbind()
{
    ((void (*)(any *))(BTRef::_itabp[3]))(this);
}

/*inline*/ int
dt1::f()
{
    return ((int (*)(dt1 *))(BTRef::_itabp[0]))(this);
}

/*inline*/ int
dt1::g(int n)
{
    return ((int (*)(dt1 *, int))(BTRef::_itabp[1]))(this, n);
}

/*inline*/ bool
dt1::dcbind(BTRef *_a0_, Uuid tid)
{
    return ((bool (*)(dt1 *, BTRef *, Uuid))(reinterpret_cast<void **>(BTRef::_itabp[2])[0]))(this, _a0_, tid);
}

/*inline*/ void
dt1::dbound()
{
    ((void (*)(dt1 *))(reinterpret_cast<void **>(BTRef::_itabp[2])[1]))(this);
}

/*inline*/ void
dt1::dunbound()
{
    ((void (*)(dt1 *))(reinterpret_cast<void **>(BTRef::_itabp[2])[2]))(this);
}

/*inline*/ void
dt1::dunbind()
{
    ((void (*)(dt1 *))(reinterpret_cast<void **>(BTRef::_itabp[2])[3]))(this);
}

void *dc1::_itable[] = {
    (void *)(&::any::dcbind),
    (void *)(&::any::dbound),
    (void *)(&::any::dunbound),
    (void *)(&::any::dunbind),
    (void *)(&dt1::f),
    (void *)(&dt1::g),
    reinterpret_cast<void *>(_itable)
};

/*inline*/ dc1 *
dc1::dt1::iobj()
{
    return reinterpret_cast<dc1 *>(_iobj);
}

/*inline*/ int
dc1::dt1__f()
{
    return 3;
}

/*inline*/ int
dc1::dt1__g(int n)
{
```

```
    return n**this->v;
}
/*inline*/ int
dc1::dt1::f()
{
    return iobj()->dt1__f();
}

/*inline*/ int
dc1::dt1::g(int n)
{
    return iobj()->dt1__g(n);
}

/*inline*/ int
dc1::_type::f()
{
    return ((int (*)(::dc1::_type *))(reinterpret_cast<void **>(BTRef::_itabp[0])[0]))(this);
}

/*inline*/ int
dc1::_type::g(int n)
{
    return ((int (*)(::dc1::_type *, int))(reinterpret_cast<void **>(BTRef::_itabp[0])[1]))(this, n);
}

/*inline*/ bool
dc1::_type::dcbind(BTRef *_a0_, Uuid tid)
{
    return ((bool (*)(::dc1::_type *, BTRef *, Uuid))(reinterpret_cast<void **>(BTRef::_itabp[1])[0]))(th
}

/*inline*/ void
dc1::_type::dbound()
{
    ((void (*)(::dc1::_type *))(reinterpret_cast<void **>(BTRef::_itabp[1])[1]))(this);
}

/*inline*/ void
dc1::_type::dunbound()
{
    ((void (*)(::dc1::_type *))(reinterpret_cast<void **>(BTRef::_itabp[1])[2]))(this);
}

/*inline*/ void
dc1::_type::dunbind()
{
    ((void (*)(::dc1::_type *))(reinterpret_cast<void **>(BTRef::_itabp[1])[3]))(this);
}

/*inline*/
dc1::dc1(int *p)
{
    this->v = p;
}


void
dc1::dbind(any *btrp) const
{
    btrp->_iobj = reinterpret_cast<void *>(this);
    btrp->_itabp = _itable;
}

/*inline*/
dc1::operator any() const
{
    any btr;
    dbind(&btr);
    return btr;
}

/*inline*/ bool
dc1::default__dcbind(BTRef *_a0_, Uuid tid)
{
    return static_cast< ::any>(*this).default__dcbind(_a0_, tid);
}

/*inline*/ void
```



```
dc1::default__dbound()
{
    static_cast< ::any>(*this).default__dbound();
}

/*inline*/ void
dc1::default__dunbound()
{
    static_cast< ::any>(*this).default__dunbound();
}

/*inline*/ void
dc1::default__dunbind()
{
    static_cast< ::any>(*this).default__dunbind();
}

void
dc1::dbind(::dt1 *btrp) const
{
    btrp->_iobj = reinterpret_cast<void *>(this);
    btrp->_itabp = &_itable[4];
}

/*inline*/
dc1::operator ::dt1() const
{
    ::dt1 btr;
    dbind(&btr);
    return btr;
}

int
test1(int n, /*type*/ dt1 o)
{
    int t = n*3;
    while(n--)
        t -= o.f();
    if(t != 0)
        printf("ERROR\n");
}

int
test2(int n, v1 *o)
{
    int t = n*3;
    while(n--)
        t -= o->f();
    if(t != 0)
        printf("ERROR\n");
}

int
test3(int n, /*type*/ dt1 o)
{
    int t = n*9*7;
    while(n--)
        t -= o.g(7);
    if(t != 0)
        printf("ERROR\n");
}

int
test4(int n, v1 *o)
{
    int t = n*9*7;
    while(n--)
        t -= o->g(7);
    if(t != 0)
        printf("ERROR\n");
}

long
timediff(timeval *tv0)
{
    timeval tvnow;
    gettimeofday(&tvnow, 0);
    return (tvnow.tv_sec-tv0->tv_sec)*1000000+(tvnow.tv_usec-tv0->tv_usec);
}
```

```
int
main()
{
    int x = 9;
    dc1 dc(&x);
    cv1 vc(&x);
    for(int n = 1; n < 100000; n += (n+3)/4) {
        int c = 1000000/n;
        timeval t0;
        printf("%d\t", n);
        gettimeofday(&t0, 0);
        for(int i = 0; i < c; i++)
            test1(n, dc);
        printf("%d\t", timediff(&t0));
        gettimeofday(&t0, 0);
        for(int i = 0; i < c; i++)
            test2(n, &vc);
        printf("%d\t", timediff(&t0));
        gettimeofday(&t0, 0);
        for(int i = 0; i < c; i++)
            test3(n, dc);
        printf("%d\t", timediff(&t0));
        gettimeofday(&t0, 0);
        for(int i = 0; i < c; i++)
            test4(n, &vc);
        printf("%d\n", timediff(&t0));
    }
}
```


E.2 Assembly Listings

E.2.1 *perf.s*

This is the assembly listing generated by compiling *perf.cc*.

```
    .file   "perf.cc"
    .version    "01.01"
gcc2_compiled.:
.globl _3dc1._itable
.data
    .align 4
    .type    _3dc1._itable,@object
    .size    _3dc1._itable,28
_3dc1._itable:
    .long dcbind__3anyP5BTRefG4Uuid
    .long dbound__3any
    .long dunbound__3any
    .long dunbind__3any
    .long f__Q23dc13dt1
    .long g__Q23dc13dt1i
    .long _3dc1._itable
.text
    .align 4
.globl dbind__C3dc1P3any
    .type    dbind__C3dc1P3any,@function
dbind__C3dc1P3any:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%edx
    movl 12(%ebp),%eax
    movl %edx,8(%eax)
    movl $_3dc1._itable,(%eax)
    leave
    ret
.Lfe1:
    .size    dbind__C3dc1P3any,.Lfe1-dbind__C3dc1P3any
    .align 4
.globl dbind__C3dc1P3dt1
    .type    dbind__C3dc1P3dt1,@function
dbind__C3dc1P3dt1:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%edx
    movl 12(%ebp),%eax
    movl %edx,8(%eax)
    movl $_3dc1._itable+16,(%eax)
    leave
    ret
.Lfe2:
    .size    dbind__C3dc1P3dt1,.Lfe2-dbind__C3dc1P3dt1
.section    .rodata
.LC0:
    .string "ERROR\n"
.text
    .align 4
.globl test1__FiG3dt1
    .type    test1__FiG3dt1,@function
test1__FiG3dt1:
    pushl %ebp
    movl %esp,%ebp
    pushl %edi
    pushl %esi
    pushl %ebx
    movl 8(%ebp),%ebx
    leal (%ebx,%ebx,2),%esi
    subl $1,%ebx
    jc .L53
    leal 12(%ebp),%edi
    .p2align 4,,7
.L54:
    movl 12(%ebp),%eax
    pushl %edi
    movl (%eax),%eax
    call *%eax
    addl $4,%esp
    subl %eax,%esi
    subl $1,%ebx
    jnc .L54
.L53:
    testl %esi,%esi
    je .L57
    pushl $.LC0
    call printf
```

```
.L57:
    leal -12(%ebp),%esp
    popl %ebx
    popl %esi
    popl %edi
    leave
    ret
.Lfe3:
    .size    test1__FiG3dt1,.Lfe3-test1__FiG3dt1
    .align 4
.globl test2__FiP2v1
    .type    test2__FiP2v1,@function
test2__FiP2v1:
    pushl %ebp
    movl %esp,%ebp
    pushl %edi
    pushl %esi
    pushl %ebx
    movl 8(%ebp),%ebx
    movl 12(%ebp),%edi
    leal (%ebx,%ebx,2),%esi
    subl $1,%ebx
    jc .L60
    .p2align 4,,7
.L61:
    movl (%edi),%eax
    pushl %edi
    movl 8(%eax),%eax
    call *%eax
    subl %eax,%esi
    addl $4,%esp
    subl $1,%ebx
    jnc .L61
.L60:
    testl %esi,%esi
    je .L63
    pushl $.LC0
    call printf
.L63:
    leal -12(%ebp),%esp
    popl %ebx
    popl %esi
    popl %edi
    leave
    ret
.Lfe4:
    .size    test2__FiP2v1,.Lfe4-test2__FiP2v1
    .align 4
.globl test3__FiG3dt1
    .type    test3__FiG3dt1,@function
test3__FiG3dt1:
    pushl %ebp
    movl %esp,%ebp
    pushl %edi
    pushl %esi
    pushl %ebx
    movl 8(%ebp),%ebx
    movl %ebx,%esi
    sall $6,%esi
    subl %ebx,%esi
    subl $1,%ebx
    jc .L66
    leal 12(%ebp),%edi
    .p2align 4,,7
.L67:
    movl 12(%ebp),%eax
    pushl $7
    pushl %edi
    movl 4(%eax),%eax
    call *%eax
    addl $8,%esp
    subl %eax,%esi
    subl $1,%ebx
    jnc .L67
.L66:
    testl %esi,%esi
    je .L70
    pushl $.LC0
    call printf
.L70:
    leal -12(%ebp),%esp
    popl %ebx
    popl %esi
    popl %edi
    leave
    ret
.Lfe5:
    .size    test3__FiG3dt1,.Lfe5-test3__FiG3dt1
    .align 4
.globl test4__FiP2v1
```



```
    .type    test4__FiP2v1,@function
test4__FiP2v1:
    pushl %ebp
    movl %esp,%ebp
    pushl %edi
    pushl %esi
    pushl %ebx
    movl 8(%ebp),%ebx
    movl 12(%ebp),%edi
    movl %ebx,%esi
    sall $6,%esi
    subl %ebx,%esi
    subl $1,%ebx
    jc .L73
    .p2align 4,,7
.L74:
    movl (%edi),%eax
    pushl $7
    pushl %edi
    movl 12(%eax),%eax
    call *%eax
    subl %eax,%esi
    addl $8,%esp
    subl $1,%ebx
    jnc .L74
.L73:
    testl %esi,%esi
    je .L76
    pushl $.LC0
    call printf
.L76:
    leal -12(%ebp),%esp
    popl %ebx
    popl %esi
    popl %edi
    leave
    ret
.Lfe6:
    .size    test4__FiP2v1,.Lfe6-test4__FiP2v1
    .align 4
.globl timediff__FP7timeval
    .type    timediff__FP7timeval,@function
timediff__FP7timeval:
    pushl %ebp
    movl %esp,%ebp
    subl $8,%esp
    pushl %esi
    pushl %ebx
    movl 8(%ebp),%esi
    pushl $0
    leal -8(%ebp),%ebx
    pushl %ebx
    call gettimeofday
    movl (%esi),%eax
    movl -8(%ebp),%ecx
    subl %eax,%ecx
    movl %ecx,%edx
    sall $5,%edx
    subl %ecx,%edx
    movl %edx,%eax
    sall $6,%eax
    subl %edx,%eax
    leal (%ecx,%eax,8),%eax
    sall $6,%eax
    movl 4(%esi),%edx
    movl 4(%ebx),%ebx
    subl %edx,%ebx
    movl %ebx,%edx
    addl %edx,%eax
    leal -16(%ebp),%esp
    popl %ebx
    popl %esi
    leave
    ret
.Lfe7:
    .size    timediff__FP7timeval,.Lfe7-timediff__FP7timeval
.section    .rodata
.LC1:
    .string "%d\t"
.LC2:
    .string "%d\n"
.text
    .align 4
.globl main
    .type    main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $76,%esp
    pushl %edi
```

```
    pushl %esi
    pushl %ebx
    movl $9,-44(%ebp)
    leal -44(%ebp),%eax
    movl %eax,-56(%ebp)
    movl $__vt_3cv1,-8(%ebp)
    movl %eax,-4(%ebp)
    movl $1,-60(%ebp)
    leal -52(%ebp),%edi
    movl %edi,-72(%ebp)
    .p2align 4,,7
.L88:
    movl $1000000,%eax
    cltd
    idivl -60(%ebp)
    movl %eax,-64(%ebp)
    movl -60(%ebp),%edi
    pushl %edi
    pushl $.LC1
    call printf
    pushl $0
    movl -72(%ebp),%edi
    pushl %edi
    call gettimeofday
    xorl %esi,%esi
    addl $16,%esp
    cmpl -64(%ebp),%esi
    jge .L90
    leal -56(%ebp),%edi
    movl %edi,-76(%ebp)
    .p2align 4,,7
.L92:
    leal -40(%ebp),%eax
    pushl %eax
    movl -76(%ebp),%edi
    pushl %edi
    call dbind__C3dc1P3dt1
    movl -40(%ebp),%ebx
    movl %ebx,-24(%ebp)
    movl -36(%ebp),%ecx
    movl %ecx,-20(%ebp)
    movl -32(%ebp),%edx
    movl %edx,-16(%ebp)
    movl -28(%ebp),%eax
    movl %eax,-12(%ebp)
    addl $8,%esp
    addl $-20,%esp
    movl %ebx,4(%esp)
    movl %ecx,8(%esp)
    movl %edx,12(%esp)
    movl %eax,16(%esp)
    movl -60(%ebp),%edi
    movl %edi,(%esp)
    call test1__FiG3dt1
    addl $20,%esp
    incl %esi
    cmpl -64(%ebp),%esi
    jl .L92
.L90:
    movl -72(%ebp),%edi
    pushl %edi
    call timediff__FP7timeval
    pushl %eax
    pushl $.LC1
    call printf
    pushl $0
    leal -52(%ebp),%edi
    movl %edi,-76(%ebp)
    pushl %edi
    call gettimeofday
    xorl %ebx,%ebx
    addl $20,%esp
    cmpl -64(%ebp),%ebx
    jge .L96
    leal -8(%ebp),%esi
    .p2align 4,,7
.L98:
    pushl %esi
    movl -60(%ebp),%edi
    pushl %edi
    call test2__FiP2v1
    addl $8,%esp
    incl %ebx
    cmpl -64(%ebp),%ebx
    jl .L98
.L96:
    movl -76(%ebp),%edi
    pushl %edi
    call timediff__FP7timeval
    pushl %eax
```



```
    pushl $.LC1
    call printf
    pushl $0
    leal -52(%ebp),%edi
    movl %edi,-68(%ebp)
    pushl %edi
    call gettimeofday
    xorl %esi,%esi
    addl $20,%esp
    cmpl -64(%ebp),%esi
    jge .L101
    leal -56(%ebp),%edi
    movl %edi,-76(%ebp)
    .p2align 4,,7
.L103:
    leal -24(%ebp),%eax
    pushl %eax
    movl -76(%ebp),%edi
    pushl %edi
    call dbind__C3dc1P3dt1
    movl -24(%ebp),%ebx
    movl %ebx,-40(%ebp)
    movl -20(%ebp),%ecx
    movl %ecx,-36(%ebp)
    movl -16(%ebp),%edx
    movl %edx,-32(%ebp)
    movl -12(%ebp),%eax
    movl %eax,-28(%ebp)
    addl $8,%esp
    addl $-20,%esp
    movl %ebx,4(%esp)
    movl %ecx,8(%esp)
    movl %edx,12(%esp)
    movl %eax,16(%esp)
    movl -60(%ebp),%edi
    movl %edi,(%esp)
    call test3__FiG3dt1
    addl $20,%esp
    incl %esi
    cmpl -64(%ebp),%esi
    jl .L103
.L101:
    movl -68(%ebp),%edi
    pushl %edi
    call timediff__FP7timeval
    pushl %eax
    pushl $.LC1
    call printf
    pushl $0
    leal -52(%ebp),%esi
    movl %esi,-72(%ebp)
    pushl %esi
    call gettimeofday
    xorl %ebx,%ebx
    addl $20,%esp
    cmpl -64(%ebp),%ebx
    jge .L107
    leal -8(%ebp),%edi
    movl %edi,-76(%ebp)
    .p2align 4,,7
.L109:
    movl -76(%ebp),%edi
    pushl %edi
    movl -60(%ebp),%edi
    pushl %edi
    call test4__FiP2v1
    addl $8,%esp
    incl %ebx
    cmpl -64(%ebp),%ebx
    jl .L109
.L107:
    pushl %esi
    call timediff__FP7timeval
    pushl %eax
    pushl $.LC2
    call printf
    addl $12,%esp
    movl -60(%ebp),%eax
    addl $3,%eax
    jns .L111
    movl -60(%ebp),%eax
    addl $6,%eax
.L111:
    sarl $2,%eax
    addl %eax,-60(%ebp)
    cmpl $99999,-60(%ebp)
    jle .L88
    xorl %eax,%eax
    leal -88(%ebp),%esp
    popl %ebx
```

```
    popl %esi
    popl %edi
    leave
    ret
.Lfe8:
    .size    main,.Lfe8-main
    .weak   __vt_3cv1
.section    .gnu.linkonce.d.__vt_3cv1,"aw",@progbits
    .align 4
    .type    __vt_3cv1,@object
    .size    __vt_3cv1,16
__vt_3cv1:
    .long 0
    .long 0
    .long f__3cv1
    .long g__3cv1i
.section    .gnu.linkonce.t.g__Q23dc13dt1i,"ax",@progbits
    .align 4
    .weak   g__Q23dc13dt1i
    .type    g__Q23dc13dt1i,@function
g__Q23dc13dt1i:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    pushl 12(%ebp)
    pushl 8(%eax)
    call dt1__g__3dc1i
    leave
    ret
.Lfe9:
    .size    g__Q23dc13dt1i,.Lfe9-g__Q23dc13dt1i
.section    .gnu.linkonce.t.f__Q23dc13dt1,"ax",@progbits
    .align 4
    .weak   f__Q23dc13dt1
    .type    f__Q23dc13dt1,@function
f__Q23dc13dt1:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    pushl 8(%eax)
    call dt1__f__3dc1
    leave
    ret
.Lfe10:
    .size    f__Q23dc13dt1,.Lfe10-f__Q23dc13dt1
.section    .gnu.linkonce.t.dunbind__3any,"ax",@progbits
    .align 4
    .weak   dunbind__3any
    .type    dunbind__3any,@function
dunbind__3any:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%edx
    pushl %eax
    movl 12(%edx),%eax
    call *%eax
    leave
    ret
.Lfe11:
    .size    dunbind__3any,.Lfe11-dunbind__3any
.section    .gnu.linkonce.t.dunbound__3any,"ax",@progbits
    .align 4
    .weak   dunbound__3any
    .type    dunbound__3any,@function
dunbound__3any:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%edx
    pushl %eax
    movl 8(%edx),%eax
    call *%eax
    leave
    ret
.Lfe12:
    .size    dunbound__3any,.Lfe12-dunbound__3any
.section    .gnu.linkonce.t.dbound__3any,"ax",@progbits
    .align 4
    .weak   dbound__3any
    .type    dbound__3any,@function
dbound__3any:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%edx
    pushl %eax
    movl 4(%edx),%eax
    call *%eax
    leave
```



```
    ret
.Lfe13:
    .size    dbound__3any,.Lfe13-dbound__3any
.section    .gnu.linkonce.t.dcbind__3anyP5BTRefG4Uuid,"ax",@progbits
    .align 4
    .weak   dcbind__3anyP5BTRefG4Uuid
    .type    dcbind__3anyP5BTRefG4Uuid,@function
dcbind__3anyP5BTRefG4Uuid:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%edx
    pushl 16(%ebp)
    pushl 12(%ebp)
    pushl %eax
    movl (%edx),%eax
    call *%eax
    leave
    ret
.Lfe14:
    .size    dcbind__3anyP5BTRefG4Uuid,.Lfe14-dcbind__3anyP5BTRefG4Uuid
.section    .gnu.linkonce.t.g__3cv1i,"ax",@progbits
    .align 4
    .weak   g__3cv1i
    .type    g__3cv1i,@function
g__3cv1i:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 4(%eax),%eax
    movl 12(%ebp),%edx
    imull (%eax),%edx
    movl %edx,%eax
    leave
    ret
.Lfe15:
    .size    g__3cv1i,.Lfe15-g__3cv1i
.section    .gnu.linkonce.t.f__3cv1,"ax",@progbits
    .align 4
    .weak   f__3cv1
    .type    f__3cv1,@function
f__3cv1:
    pushl %ebp
    movl %esp,%ebp
    movl $3,%eax
    leave
    ret
.Lfe16:
    .size    f__3cv1,.Lfe16-f__3cv1
.section    .gnu.linkonce.t.dt1__g__3dc1i,"ax",@progbits
    .align 4
    .weak   dt1__g__3dc1i
    .type    dt1__g__3dc1i,@function
dt1__g__3dc1i:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%eax
    movl 12(%ebp),%edx
    imull (%eax),%edx
    movl %edx,%eax
    leave
    ret
.Lfe17:
    .size    dt1__g__3dc1i,.Lfe17-dt1__g__3dc1i
.section    .gnu.linkonce.t.dt1__f__3dc1,"ax",@progbits
    .align 4
    .weak   dt1__f__3dc1
    .type    dt1__f__3dc1,@function
dt1__f__3dc1:
    pushl %ebp
    movl %esp,%ebp
    movl $3,%eax
    leave
    ret
.Lfe18:
    .size    dt1__f__3dc1,.Lfe18-dt1__f__3dc1
    .ident  "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

E.2.2 *perf2.s*

This is the assembly listing generated by compiling *perf2.cc*.

```
    .file   "perf2.cc"
    .version    "01.01"
gcc2_compiled.:
.globl _3dc1._itable
.data
    .align 4
    .type    _3dc1._itable,@object
    .size    _3dc1._itable,28
_3dc1._itable:
    .long dcbind__3anyP5BTRefG4Uuid
    .long dbound__3any
    .long dunbound__3any
    .long dunbind__3any
    .long f__Q23dc13dt1
    .long g__Q23dc13dt1i
    .long _3dc1._itable
.text
    .align 4
.globl dbind__C3dc1P3any
    .type    dbind__C3dc1P3any,@function
dbind__C3dc1P3any:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%edx
    movl 12(%ebp),%eax
    movl %edx,8(%eax)
    movl $_3dc1._itable,(%eax)
    leave
    ret
.Lfe1:
    .size    dbind__C3dc1P3any,.Lfe1-dbind__C3dc1P3any
    .align 4
.globl dbind__C3dc1P3dt1
    .type    dbind__C3dc1P3dt1,@function
dbind__C3dc1P3dt1:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%edx
    movl 12(%ebp),%eax
    movl %edx,8(%eax)
    movl $_3dc1._itable+16,(%eax)
    leave
    ret
.Lfe2:
    .size    dbind__C3dc1P3dt1,.Lfe2-dbind__C3dc1P3dt1
.section    .rodata
.LC0:
    .string "ERROR\n"
.text
    .align 4
.globl test1__FiG3dt1
    .type    test1__FiG3dt1,@function
test1__FiG3dt1:
    pushl %ebp
    movl %esp,%ebp
    pushl %edi
    pushl %esi
    pushl %ebx
    movl 8(%ebp),%ebx
    leal (%ebx,%ebx,2),%esi
    subl $1,%ebx
    jc .L55
    leal 12(%ebp),%edi
    .p2align 4,,7
.L56:
    movl 12(%ebp),%eax
    pushl %edi
    movl (%eax),%eax
    call *%eax
    addl $4,%esp
    subl %eax,%esi
    subl $1,%ebx
    jnc .L56
.L55:
    testl %esi,%esi
    je .L59
    pushl $.LC0
    call printf
.L59:
    leal -12(%ebp),%esp
    popl %ebx
    popl %esi
    popl %edi
```



```
    leave
    ret
.Lfe3:
    .size    test1__FiG3dt1,.Lfe3-test1__FiG3dt1
    .align 4
.globl test2__FiP2v1
    .type    test2__FiP2v1,@function
test2__FiP2v1:
    pushl %ebp
    movl %esp,%ebp
    pushl %edi
    pushl %esi
    pushl %ebx
    movl 8(%ebp),%ebx
    movl 12(%ebp),%edi
    leal (%ebx,%ebx,2),%esi
    subl $1,%ebx
    jc .L62
    .p2align 4,,7
.L63:
    movl (%edi),%eax
    pushl %edi
    movl 8(%eax),%eax
    call *%eax
    subl %eax,%esi
    addl $4,%esp
    subl $1,%ebx
    jnc .L63
.L62:
    testl %esi,%esi
    je .L65
    pushl $.LC0
    call printf
.L65:
    leal -12(%ebp),%esp
    popl %ebx
    popl %esi
    popl %edi
    leave
    ret
.Lfe4:
    .size    test2__FiP2v1,.Lfe4-test2__FiP2v1
    .align 4
.globl test3__FiG3dt1
    .type    test3__FiG3dt1,@function
test3__FiG3dt1:
    pushl %ebp
    movl %esp,%ebp
    pushl %edi
    pushl %esi
    pushl %ebx
    movl 8(%ebp),%ebx
    movl %ebx,%esi
    sall $6,%esi
    subl %ebx,%esi
    subl $1,%ebx
    jc .L68
    leal 12(%ebp),%edi
    .p2align 4,,7
.L69:
    movl 12(%ebp),%eax
    pushl $7
    pushl %edi
    movl 4(%eax),%eax
    call *%eax
    addl $8,%esp
    subl %eax,%esi
    subl $1,%ebx
    jnc .L69
.L68:
    testl %esi,%esi
    je .L72
    pushl $.LC0
    call printf
.L72:
    leal -12(%ebp),%esp
    popl %ebx
    popl %esi
    popl %edi
    leave
    ret
.Lfe5:
    .size    test3__FiG3dt1,.Lfe5-test3__FiG3dt1
    .align 4
.globl test4__FiP2v1
    .type    test4__FiP2v1,@function
test4__FiP2v1:
    pushl %ebp
    movl %esp,%ebp
    pushl %edi
```

```
    pushl %esi
    pushl %ebx
    movl 8(%ebp),%ebx
    movl 12(%ebp),%edi
    movl %ebx,%esi
    sall $6,%esi
    subl %ebx,%esi
    subl $1,%ebx
    jc .L75
    .p2align 4,,7
.L76:
    movl (%edi),%eax
    pushl $7
    pushl %edi
    movl 12(%eax),%eax
    call *%eax
    subl %eax,%esi
    addl $8,%esp
    subl $1,%ebx
    jnc .L76
.L75:
    testl %esi,%esi
    je .L78
    pushl $.LC0
    call printf
.L78:
    leal -12(%ebp),%esp
    popl %ebx
    popl %esi
    popl %edi
    leave
    ret
.Lfe6:
    .size    test4__FiP2v1,.Lfe6-test4__FiP2v1
    .align 4
.globl timediff__FP7timeval
    .type    timediff__FP7timeval,@function
timediff__FP7timeval:
    pushl %ebp
    movl %esp,%ebp
    subl $8,%esp
    pushl %esi
    pushl %ebx
    movl 8(%ebp),%esi
    pushl $0
    leal -8(%ebp),%ebx
    pushl %ebx
    call gettimeofday
    movl (%esi),%eax
    movl -8(%ebp),%ecx
    subl %eax,%ecx
    movl %ecx,%edx
    sall $5,%edx
    subl %ecx,%edx
    movl %edx,%eax
    sall $6,%eax
    subl %edx,%eax
    leal (%ecx,%eax,8),%eax
    sall $6,%eax
    movl 4(%esi),%edx
    movl 4(%ebx),%ebx
    subl %edx,%ebx
    movl %ebx,%edx
    addl %edx,%eax
    leal -16(%ebp),%esp
    popl %ebx
    popl %esi
    leave
    ret
.Lfe7:
    .size    timediff__FP7timeval,.Lfe7-timediff__FP7timeval
.section    .rodata
.LC1:
    .string "%d\t"
.LC2:
    .string "%d\n"
.text
    .align 4
.globl main
    .type    main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $76,%esp
    pushl %edi
    pushl %esi
    pushl %ebx
    movl $9,-44(%ebp)
    leal -44(%ebp),%eax
    movl %eax,-56(%ebp)
```



```
    movl $__vt_3cv1,-8(%ebp)
    movl %eax,-4(%ebp)
    movl $1,-60(%ebp)
    leal -52(%ebp),%edi
    movl %edi,-72(%ebp)
    .p2align 4,,7
.L90:
    movl $1000000,%eax
    cltd
    idivl -60(%ebp)
    movl %eax,-64(%ebp)
    movl -60(%ebp),%edi
    pushl %edi
    pushl $.LC1
    call printf
    pushl $0
    movl -72(%ebp),%edi
    pushl %edi
    call gettimeofday
    xorl %esi,%esi
    addl $16,%esp
    cmpl -64(%ebp),%esi
    jge .L92
    leal -56(%ebp),%edi
    movl %edi,-76(%ebp)
    .p2align 4,,7
.L94:
    leal -40(%ebp),%eax
    pushl %eax
    movl -76(%ebp),%edi
    pushl %edi
    call dbind__C3dc1P3dt1
    movl -40(%ebp),%ebx
    movl %ebx,-24(%ebp)
    movl -36(%ebp),%ecx
    movl %ecx,-20(%ebp)
    movl -32(%ebp),%edx
    movl %edx,-16(%ebp)
    movl -28(%ebp),%eax
    movl %eax,-12(%ebp)
    addl $8,%esp
    addl $-20,%esp
    movl %ebx,4(%esp)
    movl %ecx,8(%esp)
    movl %edx,12(%esp)
    movl %eax,16(%esp)
    movl -60(%ebp),%edi
    movl %edi,(%esp)
    call test1__FiG3dt1
    addl $20,%esp
    incl %esi
    cmpl -64(%ebp),%esi
    jl .L94
.L92:
    movl -72(%ebp),%edi
    pushl %edi
    call timediff__FP7timeval
    pushl %eax
    pushl $.LC1
    call printf
    pushl $0
    leal -52(%ebp),%edi
    movl %edi,-76(%ebp)
    pushl %edi
    call gettimeofday
    xorl %ebx,%ebx
    addl $20,%esp
    cmpl -64(%ebp),%ebx
    jge .L98
    leal -8(%ebp),%esi
    .p2align 4,,7
.L100:
    pushl %esi
    movl -60(%ebp),%edi
    pushl %edi
    call test2__FiP2v1
    addl $8,%esp
    incl %ebx
    cmpl -64(%ebp),%ebx
    jl .L100
.L98:
    movl -76(%ebp),%edi
    pushl %edi
    call timediff__FP7timeval
    pushl %eax
    pushl $.LC1
    call printf
    pushl $0
    leal -52(%ebp),%edi
    movl %edi,-68(%ebp)
```

```
    pushl %edi
    call gettimeofday
    xorl %esi,%esi
    addl $20,%esp
    cmpl -64(%ebp),%esi
    jge .L103
    leal -56(%ebp),%edi
    movl %edi,-76(%ebp)
    .p2align 4,,7
.L105:
    leal -24(%ebp),%eax
    pushl %eax
    movl -76(%ebp),%edi
    pushl %edi
    call dbind__C3dc1P3dt1
    movl -24(%ebp),%ebx
    movl %ebx,-40(%ebp)
    movl -20(%ebp),%ecx
    movl %ecx,-36(%ebp)
    movl -16(%ebp),%edx
    movl %edx,-32(%ebp)
    movl -12(%ebp),%eax
    movl %eax,-28(%ebp)
    addl $8,%esp
    addl $-20,%esp
    movl %ebx,4(%esp)
    movl %ecx,8(%esp)
    movl %edx,12(%esp)
    movl %eax,16(%esp)
    movl -60(%ebp),%edi
    movl %edi,(%esp)
    call test3__FiG3dt1
    addl $20,%esp
    incl %esi
    cmpl -64(%ebp),%esi
    jl .L105
.L103:
    movl -68(%ebp),%edi
    pushl %edi
    call timediff__FP7timeval
    pushl %eax
    pushl $.LC1
    call printf
    pushl $0
    leal -52(%ebp),%esi
    movl %esi,-72(%ebp)
    pushl %esi
    call gettimeofday
    xorl %ebx,%ebx
    addl $20,%esp
    cmpl -64(%ebp),%ebx
    jge .L109
    leal -8(%ebp),%edi
    movl %edi,-76(%ebp)
    .p2align 4,,7
.L111:
    movl -76(%ebp),%edi
    pushl %edi
    movl -60(%ebp),%edi
    pushl %edi
    call test4__FiP2v1
    addl $8,%esp
    incl %ebx
    cmpl -64(%ebp),%ebx
    jl .L111
.L109:
    pushl %esi
    call timediff__FP7timeval
    pushl %eax
    pushl $.LC2
    call printf
    addl $12,%esp
    movl -60(%ebp),%eax
    addl $3,%eax
    jns .L113
    movl -60(%ebp),%eax
    addl $6,%eax
.L113:
    sarl $2,%eax
    addl %eax,-60(%ebp)
    cmpl $99999,-60(%ebp)
    jle .L90
    xorl %eax,%eax
    leal -88(%ebp),%esp
    popl %ebx
    popl %esi
    popl %edi
    leave
    ret
.Lfe8:
```



```
    .size    main,.Lfe8-main
    .weak   __vt_3cv1
.section    .gnu.linkonce.d.__vt_3cv1,"aw",@progbits
    .align 4
    .type    __vt_3cv1,@object
    .size    __vt_3cv1,16
__vt_3cv1:
    .long 0
    .long 0
    .long f__3cv1
    .long g__3cv1i
.section    .gnu.linkonce.t.g__Q23dc13dt1i,"ax",@progbits
    .align 4
    .weak   g__Q23dc13dt1i
    .type    g__Q23dc13dt1i,@function
g__Q23dc13dt1i:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 8(%eax),%eax
    movl (%eax),%eax
    movl 12(%ebp),%edx
    imull (%eax),%edx
    movl %edx,%eax
    leave
    ret
.Lfe9:
    .size    g__Q23dc13dt1i,.Lfe9-g__Q23dc13dt1i
.section    .gnu.linkonce.t.f__Q23dc13dt1,"ax",@progbits
    .align 4
    .weak   f__Q23dc13dt1
    .type    f__Q23dc13dt1,@function
f__Q23dc13dt1:
    pushl %ebp
    movl %esp,%ebp
    movl $3,%eax
    leave
    ret
.Lfe10:
    .size    f__Q23dc13dt1,.Lfe10-f__Q23dc13dt1
.section    .gnu.linkonce.t.dunbind__3any,"ax",@progbits
    .align 4
    .weak   dunbind__3any
    .type    dunbind__3any,@function
dunbind__3any:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%edx
    pushl %eax
    movl 12(%edx),%eax
    call *%eax
    leave
    ret
.Lfe11:
    .size    dunbind__3any,.Lfe11-dunbind__3any
.section    .gnu.linkonce.t.dunbound__3any,"ax",@progbits
    .align 4
    .weak   dunbound__3any
    .type    dunbound__3any,@function
dunbound__3any:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%edx
    pushl %eax
    movl 8(%edx),%eax
    call *%eax
    leave
    ret
.Lfe12:
    .size    dunbound__3any,.Lfe12-dunbound__3any
.section    .gnu.linkonce.t.dbound__3any,"ax",@progbits
    .align 4
    .weak   dbound__3any
    .type    dbound__3any,@function
dbound__3any:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%edx
    pushl %eax
    movl 4(%edx),%eax
    call *%eax
    leave
    ret
.Lfe13:
    .size    dbound__3any,.Lfe13-dbound__3any
.section    .gnu.linkonce.t.dcbind__3anyP5BTRefG4Uuid,"ax",@progbits
    .align 4
```

```
    .weak   dcbind__3anyP5BTRefG4Uuid
    .type    dcbind__3anyP5BTRefG4Uuid,@function
dcbind__3anyP5BTRefG4Uuid:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%edx
    pushl 16(%ebp)
    pushl 12(%ebp)
    pushl %eax
    movl (%edx),%eax
    call *%eax
    leave
    ret
.Lfe14:
    .size    dcbind__3anyP5BTRefG4Uuid,.Lfe14-dcbind__3anyP5BTRefG4Uuid
.section    .gnu.linkonce.t.g__3cv1i,"ax",@progbits
    .align 4
    .weak   g__3cv1i
    .type    g__3cv1i,@function
g__3cv1i:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 4(%eax),%eax
    movl 12(%ebp),%edx
    imull (%eax),%edx
    movl %edx,%eax
    leave
    ret
.Lfe15:
    .size    g__3cv1i,.Lfe15-g__3cv1i
.section    .gnu.linkonce.t.f__3cv1,"ax",@progbits
    .align 4
    .weak   f__3cv1
    .type    f__3cv1,@function
f__3cv1:
    pushl %ebp
    movl %esp,%ebp
    movl $3,%eax
    leave
    ret
.Lfe16:
    .size    f__3cv1,.Lfe16-f__3cv1
    .ident  "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```


E.3 Results

E.3.1 *Raw performance data from* `perf.cc`

1	202207	91753	215405	110356
2	127605	59997	149950	77833
3	108218	49920	124103	65227
4	91794	46882	113857	59885
5	95157	51515	117210	66972
7	84450	46868	104675	59989
9	78513	42154	97745	57568
12	73117	39608	92259	53201
15	70638	38412	88410	52436
19	67643	36895	85759	50389
24	65501	36091	82836	49175
30	63519	35058	81077	47899
38	62877	34881	79789	47548
48	62966	33279	78514	46659
60	61577	33038	77656	46266
75	60427	33456	77014	47304
94	59474	32953	76417	45797
118	59685	32796	75375	46389
148	58698	32022	75618	45364
185	58487	32935	75648	45456
232	58444	31529	75106	45124
290	58314	33011	74873	45111
363	58352	32745	74085	45841
454	57215	32934	74542	44982
568	57959	32129	73621	45794
710	57046	33049	74364	45542
888	57599	32838	73749	45620
1110	57182	32057	74221	44766
1388	57792	31974	74221	44778
1735	58265	32570	74236	44951
2169	57731	32555	74250	44744
2712	57387	33167	74241	44921
3390	57545	31573	73977	44446
4238	57437	31925	73084	45379
5298	56573	32579	73935	44576
6623	57195	31734	73780	44632
8279	57021	32858	74182	44735
10349	57269	31238	73744	44640
12937	57397	32439	73069	45854
16172	56039	31683	73226	44799
20215	56872	31638	73509	44464
25269	56590	32150	73016	44696
31587	55647	31792	72681	44002
39484	56957	30836	73195	44243
49355	56989	30917	73230	44257
61694	57044	31909	72383	45040
77118	52805	30084	68288	41609
96398	55709	31087	70733	43196

E.3.2 *Raw performance data from* `perf2.cc`

1	178615	91774	221749	110517
2	101291	59713	116808	78640
3	88371	50718	111772	67232
4	65811	46212	88150	59915
5	66944	51409	92016	68207
7	59147	46025	82613	60191
9	52920	42146	74718	57254
12	48054	39651	67317	53575
15	44183	38295	66202	52610
19	41609	36770	62217	50434
24	39606	35805	60002	49220
30	38294	34490	58062	48675
38	36454	34880	57476	47964
48	35990	33871	55748	46705
60	35422	33968	54257	47183
75	33629	33555	54582	46055
94	34004	32246	54079	45758
118	33552	32892	52825	46144
148	32654	32575	52372	46043
185	32443	32897	52780	45490
232	33070	31790	52557	45152
290	32633	32199	52380	45034
363	32475	32189	52333	45261
454	32363	32932	51692	44902
568	32669	32133	51509	45057
710	32243	32859	51478	45699
888	31660	32843	51405	45656
1110	31292	32053	51356	45580
1388	31253	32842	51244	45672
1735	31241	32550	51215	45858
2169	31217	33909	51973	44816
2712	31183	32654	51973	44629
3390	31122	33133	52239	44452
4238	31897	31058	51988	44563
5298	31049	32541	51758	44588
6623	31109	32361	51579	44699
8279	30959	32881	51958	44639
10349	31824	31254	51759	44664
12937	31050	32410	51720	45018
16172	30735	31675	51286	44130
20215	30855	32172	51441	44490
25269	30720	32131	50377	44951
31587	30526	31853	50087	44792
39484	30747	31726	50331	44875
49355	31624	31728	50343	44649
61694	31559	31818	50404	44180
77118	29769	29371	47931	41554
96398	30273	31025	49150	43368

F

SHACC Manual Page

NAME

shacc − Shaggy's Homebrew Alternative Compiler Compiler (v0.5)

SYNOPSIS

shacc [**−d**] [**−e**] [**−t**] [**−i**] [**−y**] [**−V**] [**−b** *prefix*] [**−p** *parser*] [**−o** *outfile*] [**−h** *hdrfile*] [*infile*|**−**]

DESCRIPTION

Shacc *(Shaggy's Homebrew Alternative Compiler Compiler)* reads an extended **yacc**-style grammar specification and generates an LR(k) parser for it, as a set of tables and code in either C or C++. The tables currently produced by **shacc** are a raw representation of the grammar, without a significant amount of processing, and thus the parsing code must do a significant amount of work. This makes **shacc** much less efficient than **yacc**, but since it has an LR(k) parser as opposed to **yacc**'s LALR(1) parser, this is currently acceptable. Future versions of **shacc** will do more work ahead of time, and use PDA's where possible, hopefully approximating the efficiency of **yacc** for the parts of the grammar that are LR(1).

OPTIONS

Options may be supplied in any order, but should come before any input files. Options should each be specified separately.

−d Generate a file, *basename*.tab.h, with `#define` statements for each token from the grammar file.

−e Generate a file, *basename*.tab.h, with a line of the form `TOKEN_NAME = number,` for each token in the grammar file. This file is suitable for #include'ing in an C/C++ enum declaration.

−t Turn on the YYDEBUG macro in the output file, to include debugging capability in the parser. Note that the yydebug variable still needs to be turned on to actually enable debugging.

−i Impure mode; disable the pure-parsing feature.

−y Enter **yacc**-compatibility mode. This disables the pure-parser feature, and sets the *basename* used as a filename prefix to just 'y', causing the output file to default to 'y.tab.c' and the token definition file to be 'y.tab.h'. This option also defines the YYYACC macro in the output.

−V Print version information about **shacc**.

−b *prefix*

Change *basename* to the given prefix. Normally, *basename* is set to the basename of the input file.

−p *parser*

Select a different source file to use as the parser engine.

−o *outfile*

Put the final output into the specified file, overriding the defaults for the output file naming. Normally, the output goes into *basename*.tab.c

−h *hdrfile*

Put the header file output (if enabled via '-d' or '-e'), into the specified output file. Normally, the header file output goes into *basename*.tab.h

FEATURES

- Shacc can handle LR(k) grammars which require an arbitrary number of lookahead tokens before performing a reduction.
- The resulting parser can used without modification in a C++ class.
- The header file (produced via the '-d' or '-e' options) will only be updated if it has changed. This prevents timestamp based tools, such as **make**, from needlessly rebuilding everything that depends upon the header if it hasn't really changed.
- Different parsers produced by **shacc** can co-exist, and multiple copies of the same parser can exist. The parser code is re-entrant as well.
- Reduction precedences can be used to give one production priority over others that reduce to the same non-terminal, thus resolving ambiguities in the grammar.
- Rules whose actions are identical will share a single action in the resulting switch() statement in the final code, reducing the overall size of the parser. It should be noted that 'identical' in this sense refers to precise character-by-character equivalence, not counting leading or trailing whitespace. In future versions, a C/C++ lexical analyzer will probably be used so only lexical token equivalence will be needed to collapse two or more actions into one.
- Rules with only one symbol (terminal or non-terminal) and no action specified will be considered unnecessary to execute, since they leave the semantic stack in the same state it is currently in. They will still be followed, but by considering them empty, more reduction paths can be considered equivalent, and can therefore be executed earlier, rather than waiting for full disambiguation.

DIFFERENCES FROM YACC

Some extensions to the standard **yacc** grammar are available, including pure parsing and reduction rule precedence.

The syntax extensions are as follows:

%pure_parser

Putting this declaration in the top portion of the file will turn on pure parsing. Pure parsing is normally enabled by default, and only disabled by the '-i' and '-y' options. The main purpose of this declaration is to provide compatibility with grammars written for other **yacc** variants, such as GNU **bison**, which require this option to enable pure parsing.

Pure parsing makes yylval no longer a global variable (which was considered 'impure'). Instead, the yylex() function is now called with the address of a local yylval, which is internal to the parser.

%rprec *number*

This applies the given number as a reduction precedence for a rule. If multiple reductions can occur to the same rule, and some of them have precedence values, only the one with the lowest precedence value will be kept. This can be used to resolve ambiguities in the grammar.

%uniq This tags a reduction that must be unique, meaning that no other active shifts or reductions can be happening at the same time. If the grammar is such that this isn't true, an 'ambuiguous grammar' error will be reported. The final implicit rule, which shifts and reduces the EOF token, is implicitly tagged as %uniq. If a rule is required to be unique, and ambiguities can still result, then the rule or some of its dependencies will have to have %rprec reduction precedences applied to them.

DIAGNOSTICS

Shacc reports the number of unreferenced tokens, non-terminals, and rules, with each report presented as a warning. If any of these quantities is zero, no warning is issued for it. Other warnings and errors may be given about the grammar, and output processing will be terminated if there are any errors.

MACROS

The following macros can be defined in the top section of the grammar file to affect the final parser:

YYSTYPE
The type used for semantic values (yylval). This defaults to **int**.

YYPARSE
The name of the main function to be called to perform the parsing. This is the main function that **shacc** will generate. This defaults to **yyparse**.

YYLEX
The lexical analyzer function to call to get new tokens. Unless pure-parser mode is inhibited, this should take the address of a YYSTYPE, which it will modify, and it should return a token value. This defaults to **yylex**.

YYDEBUGFLAG
The name of the flag used to enable/disable debugging. This defaults to *yydebug*.

YYPARSE_VARS
Any additional local variables needed by the parser. These will be made available to the action routines. This should be defined to be any actual declaration sequence for the required variables.

YYPARSE_PARAMS
Any additional parameters the parse function should accept. This should be the actual parameter declaration sequence needed.

YYMALLOC
A memory-allocation function. This defaults to *malloc*.

YYFREE
A memory-freeing function. This defaults to *free*.

YYIMPURE
If defined, this disables pure-parsing. This is usually better accomplished by using the '-i' option or by enabling yacc compatibility mode with '-y'.

YYEPRINTF
A printf-style error reporting function. This function should take its format string, and variable number of arguments, and print the text as-is to the standard error output. This defaults to *printf*. Note that a simple fprintf to standard error requires an intermediate function which calls vfprintf, since additional arguments cannot be specified with this macro.

YYERROR
A function for reporting formatted errors. By default, this is *yyerror*. Calls to this function generally do not include newlines, as it is expected that YYERROR will format the string, usually including filename and line number information, as well as a trailing new-

line.

USING SHACC FOR C++

To use **shacc** to generate C++ code which can be contained within a class, there are several options. One way is to write an alternative parser engine, wrapped in a C++ class, and use the '-p' option to select this. Another way is to use the standard parser engine, which already supports some C++ functionality when certain macros are enabled.

To enable C++ mode using the standard parser engine, the following macros should be defined, in addition to setting YYPARSE and YYLEX appropriately:

YYCLASS
The enclosing class for the parser. It should be noted that any members of this class that need to be accessible to any actions should be made protected or public, since the parser needs to derive a new base class from YYCLASS. YYPARSE is placed in the class YYCLASS.

In addition, the '-o' option should probably be used to change the output filename to something more suitable for C++.

ALTERNATIVE PARSER ENGINES

The standard parser engine comes from a file normally called shacc.c.parser, although this can be changed when **shacc** is built. Using the '-p' option, it is possible to select a different parser engine. It would probably be wise to use shacc.c.parser as an example for this, due to the limited documentation here. In the final output, **shacc** will replace the following strings with tables and other information from the grammar:

$ACTION$
switch statement for all actions to be performed

$RULES$
reduction rules, with their associated actions

$SYMBOLS$
the symbol array, referenced by the rules to get a series of symbols (tokens or non-terminals) to match for a reduction

$NONTERMS$
indices into the rule array for each non-terminal

$NAMES$
names for tokens and non-terminals (only if debugging is enabled)

$LIMITS$
values, in enum-ready form, for YYNRULES, YYNNONTERMS, YYNTOKENS, YYNSYMS, and YYINITDEPTH

$TOKENENUMS$
the equivalent to the output from the '-e' option

$TOKENDEFS$
the equivalent to the output from the '-d' option

$TOKENPREC$
precedence values for tokens

INPUT FILES

file.y yacc/shacc grammar description

OUTPUT FILES

file.tab.c C/C++ parser and rules
file.tab.h C/C++ tokens (in #define or enum form)

OTHER FILES

/usr/local/lib/shacc.c.parser default parser engine; this may be in a different place depending upon how **shacc** was built

SHORTCOMINGS

Standard **yacc** precedence rules don't work yet. For the most part, they are unnecessary, since the `%rprec` tag is more powerful, and normal precedence can be simulated by rewriting the rules, leading to a clearer grammar. At some point in the future, they will probably be added, but with a warning, since they can often hide subtle errors in the grammar.

Also, the %union construct is not currently supported. Since YYSTYPE can be defined to be any type, including a union, it is not really essential. The convenience of tagging tokens with with parts of the union to use is not available either, but explicitly stating the union member being accessed in each rule can avoid some subtle errors anyway. For the sake of completeness, and **yacc** compatibility, %union will probably be added in a future version of **shacc**.

BUGS

There are probably a few floating about, but none that I currently know of. All of the information stated above ought to be correct, but I haven't had time to thoroughly test everything, so there may be problems. Feel free to mail the author to report any problems.

SOURCE

This program, along with its full source code, is available from *http://www.csh.rit.edu/shaggy/software.html* and *ftp://ftp.csh.rit.edu/pub/members/shaggy/shagware*.

SEE ALSO

yacc(1), **bison(1)**.

AUTHOR

Shacc was written by Frank Barrus <*shaggy@csh.rit.edu*>, as a necessary part of completeting his R.I.T. Masters Thesis, *Dynamic Encapsulation of C++ Objects for Distributed Object-Oriented Systems.*

Glossary

It should be noted that some of these definitions are specific to the DECO object framework, or at least biased towards it, and thus they may differ somewhat from the standard technical meanings that these words might have in other contexts.

abstraction
: a form of "selective ignorance", whereby an entity is considered only in terms of specific features, so that the state, presence, or absence of any other features do not matter.

argument
: a value passed to a function. This term is used exclusively for values, to distinguish them from *parameters*. **Formal arguments** are the variables that are bound to the *actual arguments* when a function is invoked. See *parameter*.

arity
: *of a function:* the number of formal arguments or parameters

attribute
: any of a number of uniquely named observable values or references that an object has. Attributes are conceptually similar to C++ public instance variables within an object; however, there is not necessarily a direct correlation, since attributes can be **synthesized attributes** where the value is not actually stored, but instead a *get* function is used to calculate the value. It is also not required for attributes to have a corresponding `set` operation. If no such operation exists, the attribute can be considered to be *read-only*, even though there may be other ways its value can change. A *constant* attribute never changes.

call-by-reference
: a means of passing arguments into a function where a reference to the original variable is passed, so that called function can obtain or change the value in that variable. In this case, the reference to the *intensional* container for the variable is handed to the function.

call-by-sharing
: a means of passing arguments info a function where the object referred to by the original variable is passed. There is no access to the original variable, so it cannot refer to a different object. However, the object it refers to can possibly be altered, depending upon the type of the parameter it was passed to. This is actually a variation of *call-by-value* where the value is a reference to an *intensional* container.

call-by-value
: a means of passing arguments into a function where the value of each variable is passed, but the called function cannot access the original variable. The *extensional* object references are handed to the function, but no access is given to the *intensional* variable that originally contained that reference.

class
: the implementation specification used by an object. With standard C++ classes, a static implementation is available at compile time to generate the correct code for objects of that class. In DC++, dynamic classes can be created which actually create objects for the classes. These objects are accessible at runtime and provide access to the code necessary for implementing the features of an object. Most classes have *constructors* for creating new objects of that class, and some also have *destructors* for properly cleaning up when the object's token goes away. Contrast to *type*.

class variable
: a variable which belongs to a class. Changes to it affect all instances of the class, since class variables are shared.

constructor
: a special method used for building the representation of a newly created object. See also *destructor*.

contravariant
: varying in the opposite direction. For example, if *a* and *b* are contravariant, and an instance of a subtype of *a*'s type is substituted for *a*, then an instance of a *super*type of *b*'s type must be substituted for *b*.

covariant
: varying in the same direction. For example, if *a* and *b* are covariant, and an instance of a subtype of *a*'s type is substituted for *a*, then an instance of a subtype of *b*'s type must be substituted for *b*.

datatype

something that can be used to describe the storage and properties of a variable or parameter in C or C++. Even when using DC++ *types* to declare variables/parameters, a *datatype* has to be specified, since in reality the implementation still uses known storage sizes for the type interface (the `TRef`). Basically, any name that can be used in C/C++ to declare a variable is a **datatype**. In an abstract sense, *classes* can be used as datatypes, while true *types* (referring to interfaces) cannot, since by definition they have no implementation and no data associated directly with them. However, since C and C++ only allow declaring variables using datatypes, the DC++ compiler provides a concrete implementation of the type interface. This allows the type interface to be used as a datatype, and thus for declaring variables.

DC++ (Dynamic C++)

a language based on C++ with extensions for supporting dynamic types and classes. This language is currently compiled by the DECO program.

DECO (Dynamic Encapsulator of C++ Objects)

a utility for converting Dynamic C++ (DC++) code to standard C++ code. It can be used either as an IDL compiler, by converting special dynamic header files (which usually end in `.dh`) to standard header files (`.dh.h`), or as a pre-processor to a C++ compiler, thus extending the language to a full implementation of DC++.

delegation

the act of passing off an invocation to a *superobject* (referenced through a parent-link) because there is no local implementation to handle it. During delegation, the `self` pseudo-variable generally stays bound to the original invoker, although `here` will be re-bound to the delegatee. Occasionally, `self` is re-bound to the delegator, as it would be for a normal object invocation.

descriptor

a data structure or C++ class used to describe some entity. See *object descriptor*, *type descriptor*, and *class descriptor*.

destructor

a special method used to clean up the representation of an object when it goes out of scope. This cleanup may involve removing representations of objects that are referred to as well. See also *constructor*.

dynamic encapsulation

encapsulation that occurs at run-time, such that the code being executed does not need any knowledge of the internal structure of an object. Instead, the code deals with an interface that is dynamically bound to the implementation for the object. Contrast with *static encapsulation*.

dynamic inheritance

inheritance that occurs at runtime, allowing parent relationships to change. Changes in dynamic inheritance occur when *superobjects* are manipulated. *Forwarders* also affect dynamic inheritance, but in a different way, since the original object now becomes the superobject and the forwarder replaces the original.

encapsulation

the process of removing access to the internal representation of an object, and treating it as a single entity (or capsule), with a well-defined interface that is accessible. Encapsulation is a form of *abstraction*. See also *static encapsulation* and *dynamic encapsulation*.

feature

of an object: any accessible operation or attribute of the object. All functional relationships and procedures in which the object can be implicitly used as the primary argument represent features of the object. The set of all features of an object represents the complete *type signature* of the object; however subsets of the complete type signature also constitute other valid type signatures for the object.

field

a portion of the storage of an object representation reserved for values of a particular class. A field corresponds to an instance variable in C++, and may or may not be directly associated with an *attribute*. Fields store the state of an object, and can be seen as references to other objects, both mutable (*intensional*) and immutable (*extensional*).

flattening

converting the representation of an object into a raw stream of bytes, suitable for persistent storage or IPC. No translation of data occurs when flattening, only translation of pointers into a flat representation. In contrast, *packing* requires additional work to translate the data.

forwarder

an object that is used to add, remove, or modify the implementation of types that another object supports. A forwarder can be *attached* to another object at runtime, and causes all references to the original object to now refer to the forwarder, with the original object acting as the *superobject* of the forwarder.

genericity

a technique which allows the definition of a type, class or function to be parameterized such that it can be customized as needed while still providing type safety. C++ templates provide a form of genericity for classes and functions. An example of genericity or *parameterized types* is being able to define a generic type for arrays that can be specialized to create a type for an array of integers.
constrained genericity puts type constraints on the types used for parameterization, rather than allowing them to be of any type. The genericity technique of *class substitution* naturally provides constrained genericity in a more simplified style.

`here`
the object that the currently invoked method belongs to. During delegation, `here` is rebound to each object that is delegated to. Contrast with `this` and `self`.

hint
a piece of information used for a potential optimization. This could, for example, be an array index at which to start searching for some key, so that successive lookups of the same key will not need to search the entire array if nothing has changed.

IDL (Interface Definition Language)
a pseudo-language that is used to describe interfaces between programs, generally for networked software. An IDL compiler converts IDL code into native code for another language, automatically generating the extra code necessary to hide the complexities of the interface. As an example, an IDL compiler for RPC's would generate functions that handle all the details of passing data across the network in a standard format.

instance
an individual object that is of a particular *class* or *type*

instance variable
a variable which belongs to an individual instance object, and thus can be changed independently for each instance. Contrast with *class variables*.

instantiation
the process of taking a *class* and generating an *instance* of it.

invariant
unable to vary. If two entities are invariant, then there is a simple one to one functional mapping between them.

IPC (Inter Process Communication)
A means of getting data from one address space to another on the same machine. This is often accomplished via shared memory or by having the kernel maintain message queues. Address translations may be required, but no data translations should be necessary.

method
any function or procedure defined for an object, including the operations on the object, and the functions for retrieving the attributes. A method is analogous to a member function in C++. However, with DECO, the binding of a method *selector* to an implementation is performed at run-time. Multiple methods can exist with the same name, but different *signatures*. Each overloaded method will be mapped to a unique *selector*.

object
any entity that can be referred to and used as an argument to a function. Some objects are *extensional* meaning that they exist and refer to a unique entity, but never change, so they only have methods for accessing attributes. Others are *intensional*, meaning that they have a unique state and act as a container. *Intensional* objects are composed of data and a means of operating upon that data as well as accessing it. Objects have a *class* and one or more *types* which they are an instance of.

object descriptor
a descriptor used for generic objects, which can be of any class. It contains a pointer to the class descriptor for the object, and a means to locate the data for the object, which may be contained within the descriptor itself if the data is small enough.

OCF (Object Call Frame)
a data structure used to send a method information about the call that was made, including the method and arguments passed, the `self`, `here`, and `super` pseudo-variables, and any other implementation-specific information for optimization. It can also be used to keep links for unraveling the call chain when errors or exceptions occur.

OOPL (Object Oriented Programming Language)
an object-based language that also has support for classes and class inheritance.

overload
to declare a method with the same name as an existing one, but with a different *signature*. The appropriate method to call will be decided by matching the types of the arguments passed to the parameters in the signature. Each overloaded method has a unique *selector* to identify it.

override
to declare a method in a subclass, subtype, or subobject, with the same name and a compatible signature as one in a superclass, supertype, or superobject. A compatible signature need not match exactly- the arguments may vary *contravariantly* and the return types may vary *covariantly*. The new overriding method has precedence over the original method when a call is invoked on either the subobject or instance of the subtype or subclass.

packing
converting the representation of an object into a byte stream in a standard form that is suitable for transport across a network. This requires translation so that the resultant byte stream can be used on multiple machines and different architectures, so the format chosen for the packed image of the object should either be universally known, or should have associated information that describes the encoding. In contrast, this requires more effort than *flattening*.

parameter
a specification of the acceptable values for an argument to a function. Note that the parameter consists of the *type* of the formal argument, although this is often misused to refer to the value that is passed in. See *argument*.

parameterized types
see *genericity*.

pre-processor
a program which takes source files and performs conversions on portions of the text, such as macro substitution or the generation of additional code, and produces output which is suitable for use as input to a compiler or assembler.

proxy object
an object that acts as a local representation of an object that is mapped into a different address space, possibly on a different machine. The proxy object acts in nearly every way like the remote object, so that programs don't need to know where an object actually resides in order to utilize it. However, such knowledge is often made available for programs that do wish to know for optimization, debugging, or security purposes.

RPC (Remote Procedure Call)
a standard method for calling functions on other machines across the network, and for converting all of the arguments into compatible data types that can be transferred to another address space. The interface for defining these is usually built with an *IDL*.

selector
an identifier used to locate a *feature* (a method or attribute). In the current implementation of DECO, selectors are offsets, uniquely associated with each type, that are used as an index into the *itable* to find the appropriate routines to invoke. In previous implementations, selectors were strings or various forms of *UUID*s.

`self`
the original receiver of an object method invocation; the instance of the most specific subclass invoked. During delegation, `self` is generally not modified. Contrast with `this` and `here`.

semantics
of a function/procedure: the set of rules defining the behavior of a function based on its arguments, and any state information that it depends upon. This behavior includes values returned, and any changes to state information (also known as *side effects*) that execution of the function/procedure causes.

of a type: the set of rules defining the behavior for all operations on an object of that type, including all state changes made to the object, defined in terms of the results obtained from subsequently executing other methods. The semantics of a type should make no assumptions about the underlying storage representation.

of a class: the set of rules defining the exact behavior of all operations on an object of that class, including an explicit description of all state changes made to that object, as well as any other external state changes involved. The semantics of a class are the precise definitions of how the class behaves, and are included in the code for class definitions in most OOPLs.

signature
of a method or function: the specific extensional ordered set of formal input and output parameter types. The *arity* is implicitly part of the signature as well.

of a type or class: the specific extensional unordered set of bound method signatures and method names.

static encapsulation
encapsulation that occurs at compile time, with the resultant output code having knowledge of the internal structure of an object in order to implement it. Contrast with *dynamic encapsulation*.

static inheritance
inheritance that occurs at compile time, and cannot be changed at runtime. This is generally the inheritance that is used to derive *subtypes* and *subclasses*, and is the only inheritance directly supported by C++. Contrast with *dynamic inheritance* and *delegation*.

subclass
a class that represents a more specialized form of another class, known as the *superclass*. A subclass generally adds new features to another class, which it is said to inherit from, or be derived from.

subtype
a type that represents a more specific form of another type, through the addition of features, or the narrowing of return types of methods. Note that parameters to methods cannot be narrowed, but they can be widened, due to their `contravariant` nature.

superclass
a class that represents a more generalized form of another class. When deriving a class from a superclass, the new class, or *subclass*, inherits all of the features and fields of the superclass, unless they are overridden by the subclass.

superobject
an object whose features are used if the invoked object does not have the desired feature. A superobject essentially acts as a generalized parent object, and provides run-time inheritance through *delegation*, whereas *superclasses* generally provide compile-time inheritance.

supertype
a type that represents a more generalized form of another type. These are represented as the superobjects of the type, since types are also objects.

`this`
a hybrid of the `self` and `here` objects used in C++. For accessing instance variables, or calling non-virtual functions, `this` acts like `here`. However, when calling virtual functions, `this` acts like `self`. Contrast with `self` and `here`.

token
any representation used to refer to some object, whether it be an *intensional* or *extensional* object. The storage area for the token can be seen as an *intensional* object. **Direct tokens** keep the entire representation together in one place. In contrast, **indirect tokens** have pointers or references to the various parts of the complete token.

type
a general classification of the intended meaning of an object. A type specifies how an object behaves, without specifying how that object is stored or implemented. If two objects are both instances of some type, then they both have the subset of features represented by that type. The *signature* and *semantics* of a type is used to determine type compatibility. A type is analogous to the interface available to an object. (contrast to *class*)

UUID (Universally Unique IDentifier)
some form of identifier, often a large number with at least 64 or 128 bits, used to globally identify some entity. These could be used as references to specific objects, classes, types, or even as method selectors. These are generated by an algorithm that often involves some identifier of the host machine, the time, and sometimes a random number as well.

Bibliography

[Abadi96] M. Abadi and L. Cardelli, *A Theory of Objects,* Springer-Verlag, New York, NY, 1996.

[Agesen95] O. Agesen, "Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance," *Software- Practice and Experience*, vol. 25, no. 9, September 1995.

[Agha87] G. Agha and C. Hewitt, "Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming," in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 49-74, The MIT Press, Cambridge, MA, 1987.

[Aho88] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA, 1988.

[Barbier92] F. Barbier, "Object-Oriented Analysis of Systems through their Dynamical Aspects," *Journal of Object-Oriented Programming*, pp. 45-51, May, 1992.

[Baron80] R. Baron and L. Shapiro, *Data Structures and their Implementation,* Van Nostrand Reinhold Company, New York, NY, 1980.

[Barrus96] F. Barrus, "Shag/OS - An Object-Oriented MicroKernel-based Distributed Operating System - Independent Study Project Final Report," *http://www.csh.rit.edu/˜shaggy/docs/sos/isp2.ps*, Rochester Institute of Technology, May, 1996.

[Barrus92] F. Barrus, "The Practicality and Performance of a Fully-Dynamic Object-Oriented Microkernel - an Independent Study Project Proposal," *http://www.csh.rit.edu/˜shaggy/docs/sos/isp1.ps*, Rochester Institute of Technology, May, 1992.

[Barrus95] F. Barrus, "Shag/OS - A Small, Dynamic, Object-Oriented MicroKernel-based Operating System - an Independent Study Project Proposal," *http://www.csh.rit.edu/˜shaggy/docs/sos/isp2.ps*, Rochester Institute of Technology, September, 1995.

[Beck90] L. Beck, *Systems Software,* 2nd ed., Addison-Wesley, Reading, MA, 1990.

[Beech87] D. Beech, "Groundwork for an Object Database Model," in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 317-354, The MIT Press, Cambridge, MA, 1987.

[Blair91a] G. Blair, "What are Object-Oriented Systems?," in *Object-Oriented Languages, Systems and Applications*, ed. G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, pp. 108-135, Halsted Press, New York, NY, 1991.

[Blair91b] G. Blair, "Types, Abstract Data Types and Polymorphism," in *Object-Oriented Languages, Systems and Applications*, ed. G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, pp. 75-107, Halsted Press, New York, NY, 1991.

[Blair91c] G. Blair, H. Bowman, and R. Lea, "Objects, Classes and Inheritance," in *Object-Oriented Languages, Systems and Applications*, ed. G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, pp. 24-41, Halsted Press, New York, NY, 1991.

[Blair91d] G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, "Future Directions in Object-Oriented Computing," in *Object-Oriented Languages, Systems and Applications*, ed. G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, pp. 348-370, Halsted Press, New York, NY, 1991.

[Bloomer92] J. Bloomer, *Power Programming with RPC,* O'Reilly & Associates, Inc., Sebastopol, CA, 1992.

[Booch91] G. Booch, *Object Oriented Design with Applications,* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.

[Burdick90] B. Burdick, "Mob, a free OOPL based on Smalltalk," *rec.arts.int-fiction usenet posting*, Purdue University, March, 1990.

[Burdick89] B. Burdick, R. Riggs, and M. Adler, "The mob Language," *rec.arts.int-fiction usenet posting*, Carnegie-Mellon University, November, 1989.

[Cargill93] T. Cargill, "The Case Against Multiple Inheritance in C++," in *The Evolution of C++*, ed. J. Waldo, pp. 101-109, The MIT Press, Cambridge, MA, 1993.

[Chang95] W. Chang and C. Tseng, "Supporting Distributed Objects in FIFO-based Message-Passing Systems," *Journal of Object-Oriented Programming*, pp. 56-64, February 1995.

[Chen96] J. Chen and S. Lee, "The Necessary and Sufficient Conditions of Type-Safe Polymorphism," *Journal of Object-Oriented Programming*, pp. 33-42, February 1996.

[Clémençon96] C. Clémençon, B. Mukherjee, and K. Schwan, "Distributed Shared Abstractions on Multiprocessors," *IEEE Transactions on Software Engineering*, pp. 132-152, February, 1996.

[Comer87] D. Comer, *Operating System Design Volume II: Internetworking with XINU,* Prentice-Hall Inc., Englewood Cliffs, NJ, 1987.

[Cook91] S. Cook, "Programming Languages Based on Objects," in *Object-Oriented Languages, Systems and Applications*, ed. G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, pp. 136-165, Halsted Press, New York, NY, 1991.

[Cook90] W. Cook, W. Hill, and P. Canning, "Inheritance is not Subtyping," *Proceedings ACM Symposium on the Principles of Programming Languages (POPL)*, January 1990.

[Dahl87] O. Dahl, "Object-Oriented Specifications," in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 561-576, The MIT Press, Cambridge, MA, 1987.

[Daniels93] J. Daniels and S. Cook, "Strategies for Sharing Objects in Distributed Systems," *Journal of Object-Oriented Programming*, pp. 27-36, January 1993.

[Davis92] S. Davis, "C++ Objects that Change their Types," *Journal of Object-Oriented Programming*, pp. 27-32, July/August 1992.

[Dijkstra68] E. Dijkstra, "Go To Statement Considered Harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147-148, March, 1968.

[Driesen96] K. Driesen and U. Hölzle, "The Direct Cost of Virtual Function Calls in C++," *OOPSLA '96*, pp. 306-323, Department of Computer Science, University of California, 1996.

[Elmasri94] R. Elmasri and S. Navathe, *Fundamentals of Database Systems,* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[Fekete] A. Fekete, M. Kaashoek, and N. Lynch, "Implementing Sequentially Consistent Shared Objects using Broadcast and Point-To-Point Communication," ??, Department of Computer Science, University of Sydney, Australia.

[Flanagan96] D. Flanagan, *Java in a Nutshell,* O'Reilly and Associates, Inc., Sebastopol, CA, 1996.

[Ford95] B. Ford, M. Hibler, and J. Lepreau, "Using Annotated Interface Definitions to Optimize RPC," ??, Department of Computer Science, University of Utah, March 31, 1995.

[Gallagher91] J. Gallagher, ''Variations on a Theme,'' in *Object-Oriented Languages, Systems and Applications*, ed. G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, pp. 42-74, Halsted Press, New York, NY, 1991.

[Goguen87] J. Goguen and J. Meseguer, ''Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics,'' in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 417-478, The MIT Press, Cambridge, MA, 1987.

[Granston93] E. Granston and V. Russo, ''Signature-Based Polymorphism for C++,'' in *The Evolution of C++*, ed. J. Waldo, pp. 121-133, The MIT Press, Cambridge, MA, 1993.

[Grossman93] M. Grossman, ''Object I/O and Runtime Type Information Via Automatic Code Generation in C++,'' *Journal of Object-Oriented Programming*, pp. 34-42, July-August 1993.

[Haddad95] H. Haddad and K. George, ''A Survey of Method Binding and Implementation Selection in Object-Oriented Programming Languages,'' *Journal of Object-Oriented Programming*, pp. 28-41, October 1995.

[Hailpern87] B. Hailpern and V. Nguyen, ''A Model for Object-Based Inheritance,'' in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 145-164, The MIT Press, Cambridge, MA, 1987.

[Harland91] D. Harland and B. Drummond, ''REKURSIV - Object-Oriented Hardware,'' in *Object-Oriented Languages, Systems and Applications*, ed. G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, pp. 270-298, Halsted Press, New York, NY, 1991.

[Harris91] W. Harris, ''Contravariance for the Rest of Us,'' *Journal of Object-Oriented Programming*, pp. 10-18, November 1991.

[Henderson94] R. Henderson and B. Zorn, ''A Comparison of Object-oriented Programming in Four Modern Languages,'' *Software- Experience and Practice*, pp. 1077-1095, November, 1994.

[Huang92] S. Huang and D. Chen, ''Efficient Algorithms for Method Dispatch in Object-Orientented Programming Systems,'' *Journal of Object-Oriented Programming*, pp. 43-54, September 1992.

[Hutchison91] D. Hutchison and J. Walpole, ''Distributed Systems and Objects,'' in *Object-Oriented Languages, Systems and Applications*, ed. G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, pp. 223-243, Halsted Press, New York, NY, 1991.

[Johnson91] R. Johnson and J. Zweig, ''Delegation in C++,'' *Journal of Object-Oriented Programming*, pp. 31-38, November 1991.

[Joyner96] I. Joyner, ''C++? A Critique of C++ and Programming and Language Trends of the 1990s,'' *http://www.progsoc.uts.edu.au/~geldridg/cpp/cppcv3.ps.gz*, vol. 3rd ed., November, 1996.

[Kahn87] K. Kahn, E. Tribble, M. Miller, and D. Bobrow, ''Vulcan: Logical Concurrent Objects,'' in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 75-112, The MIT Press, Cambridge, MA, 1987.

[Kent89] W. Kent, ''Second Generation Object Models,'' *http://home.earthlink.net/~billkent/Doc/twogen.htm*, December, 1989.

[Kent91] W. Kent, ''A Rigorous Model of Object Reference, Identity, and Existence,'' *Journal of Object-Oriented Programming*, pp. 28-36, June 1991.

[Kent94] W. Kent, ''Non-Materialized, Materialized, and Effective Types,'' *http://home.earthlink.net/~billkent/Doc/ntypes.htm*, February, 1994.

[Kent78] W. Kent, *Data and Reality,* North-Holland Publishing Company, New York, NY, 1978.

[Kernighan79] B. Kernighan and L. Cherry, ''Typesetting Mathematics - Users's Guide (Second Edition),'' in *UNIX Programmer's Manual*, ed. Holt, Rinehart, and Winston, vol. Seventh Edition, Volume 2, pp. 146-156, Bell Telephone Laboratories, Inc., Murray Hill, NJ, 1979.

[Kesteloot96] L. Kesteloot, ''GOTOs Considered Useful,'' *http://tofu.alt.net/~lk/style/essays.html*, March, 1996.

[Koenig95a] A. Koenig, *Working Paper for Draft Proposed International Standard for Information Systems-- Programming Language C++, (X3J16/95),* AT&T Bell Laboratories, Murray Hill, NJ, April, 1995.

[Koenig96] A. Koenig, *Working Paper for Draft Proposed International Standard for Information Systems-- Programming Language C++, (X3J16/96),* AT&T Research, Murray Hill, NJ, December, 1996.

[Koenig94] A. Koenig, ''Thoughts on Abstraction,'' *Journal of Object-Oriented Programming*, pp. 68-70, October 1994.

[Koenig95b] A. Koenig, ''Another Handle Variation,'' *Journal of Object-Oriented Programming*, pp. 61-63, November-December 1995.

[Koenig95c] A. Koenig and A. Koenig, ''Variations on a Handle Theme,'' *Journal of Object-Oriented Programming*, pp. 77-80, October 1995.

[Kristensen87] B. Kristensen, O. Madsen, B. Møller-Pedersen, and K. Nygaard, ''The BETA Programming Language,'' in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 8-48, The MIT Press, Cambridge, MA, 1987.

[Lesk79a] M. Lesk, ''Updating Publication Lists,'' in *UNIX Programmer's Manual*, ed. Holt, Rinehart, and Winston, vol. Seventh Edition, Volume 2, pp. 188-195, Bell Telephone Laboratories, Inc., Murray Hill, NJ, 1979.

[Lesk79b] M. Lesk, ''Tbl - A Program to Format Tables,'' in *UNIX Programmer's Manual*, ed. Holt, Rinehart, and Winston, vol. Seventh Edition, Volume 2, pp. 157-174, Bell Telephone Laboratories, Inc., Murray Hill, NJ, 1979.

[Levine92] J. Levine, T. Mason, and D. Brown, *Lex & Yacc,* O'Reilly and Associates, Inc., Sebastopol, CA, 1992.

[Lindholm97] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification,* Addison-Wesley, Reading, MA, 1997.

[Liskov87] B. Liskov, ''Data Abstraction and Hierarchy,'' *OOPSLA '87 Addendum to the Proceedings*, pp. 17-34, October, 1987.

[Liskov81] B. Liskov, T. Bloom, J. Schaffert, R. Scheifler, R. Atkinson, A. Snyder, and E. Moss, *CLU Reference Manual,* Springer-Verlag, Berlin, Germany, 1981.

[Madsen87] O. Madsen, ''Block Structure and Object-Oriented Languages,'' in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 113-128, The MIT Press, Cambridge, MA, 1987.

[Madsen91] O. Madsen, ''Basic Principles of the BETA Programming Language,'' in *Object-Oriented Languages, Systems and Applications*, ed. G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, pp. 299-327, Halsted Press, New York, NY, 1991.

[Mariani91] J. Mariani, ''Object-Oriented Database Systems,'' in *Object-Oriented Languages, Systems and Applications*, ed. G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, pp. 166-222, Halsted Press, New York, NY, 1991.

[Martin93] B. Martin, ''The Separation of Interface and Implementation in C++,'' in *The Evolution of C++*, ed. J. Waldo, pp. 249-265, The MIT Press, Cambridge, MA, 1993.

[McKusick96] M. McKusick, K. Bostic, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.4BSD Operating System,* Addison-Wesley, Reading, MA, 1996.

[Mowbray95] T. Mowbray and R. Zahavi, *The Essential CORBA - Systems Integration Using Distributed Objects,* John Wiley & Sons, New York, NY, 1995.

[Nascimento92] C. Nascimento and J. Dollimore, ''Behavior Maintenance of Migrating Objects in a Distributed Object-Oriented Environment,'' *Journal of Object-Oriented Programming*, pp. 25-33, September 1992.

[Oliver93] I. Oliver, *Programming Classics: Implementing the World's Best Algorithms,* Prentice Hall, Englewood Cliffs, NJ, 1993.

[Orfali96] R. Orfali, D. Harkey, and J. Edwards, *The Essential Distributed Objects Survival Guide,* John Wiley & Sons, New York, NY, 1996.

[Ormsby91] A. Ormsby, ''Object-Oriented Design Methods,'' in *Object-Oriented Languages, Systems and Applications*, ed. G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, pp. 203-222, Halsted Press, New York, NY, 1991.

[Ossana79] J. Ossana, ''NROFF/TROFF User's Manual,'' in *UNIX Programmer's Manual*, ed. Holt, Rinehart, and Winston, vol. Seventh Edition, Volume 2, pp. 196-229, Bell Telephone Laboratories, Inc., Murray Hill, NJ, 1979.

[Ossher87] H. Ossher, ''A Mechanism for Specifying the Structure of Large, Layered Systems,'' in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 219-252, The MIT Press, Cambridge, MA, 1987.

[Palsberg94] J. Palsberg and M. Schwartzbach, *Object-oriented Type Systems,* John Wiley and Sons, 1994.

[Partridge94] C. Partridge, ''Modeling the Real World: Are Classes Abstractions or Objects?,'' *Journal of Object-Oriented Programming*, pp. 39-45, November-December 1994.

[Plauger95] P. Plauger, *The Draft Standard C++ Library,* Prentice Hall, Englewood Cliffs, NJ, 1995.

[Prabhakar95] E. Prabhakar, ''Implementing Distributed Objects - Doing it the easy way with NeXT's PDO,'' *Dr. Dobb's Journal*, pp. 80-85, August, 1995.

[Pratt84] T. Pratt, *Programming Languages: Design and Implementation,* 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 1984.

[Riggs89] R. Riggs, ''Forwarders in mob,'' *rec.arts.int-fiction usenet posting*, Carnegie-Mellon University, November, 1989.

[Rist95] R. Rist and R. Terwilliger, *Object-Oriented Programming in Eiffel,* Prentice-Hall, Sydney, Australia, 1995.

[Silberschatz90] A. Silberschatz, J. Peterson, P. Galvin, *Operating System Concepts,* 3rd ed., Addison-Wesley, Reading, MA, 1990.

[Skarra87] A. Skarra and S. Zdonik, ''Type Evolution in an Object-Oriented Database,'' in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 317-354, The MIT Press, Cambridge, MA, 1987.

[Snyder87] A. Snyder, ''Inheritance and the Development of Encapsulated Software Components,'' in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 165-188, The MIT Press, Cambridge, MA, 1987.

[Stroustrup93] B. Stroustrup, *The C++ Programming Language,* 2nd ed., Addison-Wesley, Reading, MA, 1993.

[Stroustrup95] B. Stroustrup, *The Design and Evolution of C++,* Addison-Wesley, Reading, MA, 1995.

[Stroustrup97] B. Stroustrup, *The C++ Programming Language,* 3rd ed., Addison-Wesley, Reading, MA, 1997.

[Tanenbaum95] A. Tanenbaum, *Distributed Operating Systems,* Prentice Hall, Englewood Cliffs, NJ, 1995.

[Tanenbaum89] A. Tanenbaum, *Computer Networks,* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[Tanenbaum92] A. Tanenbaum, M. Kaashoek, and H. Bal, ''Parallel Programming Using Shared Objects and Broadcasting,'' *Computer*, pp. 10-19, August, 1992.

[Ungar87] D. Ungar and R. Smith, ''Self: The Power of Simplicity,'' *OOPSLA '87 Proceedings*, pp. 227-242, October, 1987.

[Voss94] R. Voss, ''Time Invariant Virtual Member Function Dispatching for C++ Evolvable Classes,'' *Journal of Object-Oriented Programming*, pp. 23-33, November-December 1994.

[Waldo93a] J. Waldo, *The Evolution of C++*, The MIT Press, Cambridge, Massachusetts, 1993.

[Waldo93b] J. Waldo, ''The Case For Multiple Inheritance in C++,'' in *The Evolution of C++*, ed. J. Waldo, pp. 111-120, The MIT Press, Cambridge, MA, 1993.

[Wegner87] P. Wegner, ''The Object-Oriented Classification Paradigm,'' in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 479-560, The MIT Press, Cambridge, MA, 1987.

[Wegner92] P. Wegner, ''Dimensions of Object-Oriented Modeling,'' *Computer*, pp. 12-19, IEEE Computer Society, October 1992.

[Wieringa95] R. Wieringa, ''Combining Static and Dynamic Modeling Methods: A Comparison of Four Methods,'' *The Computer Journal*, vol. 38, no. 1, pp. 17-30, 1995.

[Wild97] F. Wild, ''The I-String Strategy - a winner in both space and time,'' *Dr. Dobb's Journal*, pp. 105-106, March, 1997.

[Wilde91] M. Wilde, ''A Dynamic C-Based Object-Oriented System for Unix,'' *IEEE Software*, pp. 73-85, May, 1991.

[Wirth95] N. Wirth, ''A Plea for Lean Software,'' *Computer*, vol. 28, no. 2, pp. 64-68, IEEE Computer Society, February 1995.

[Wolczko92] Mario Wolczko, ''Encapsulation, Delegation, and Inheritance in Object-Oriented Languages,'' *Software Engineering Journal*, pp. 95-101, March, 1992.

Cited References

Only references that have been explicitly referred to in the text are listed below.

- Chapter 0 - Introduction and Background
  [*Barrus92*] [*Barrus95*] [*Barrus96*] [*Bloomer92*] [*Joyner96*] [*Martin93*]
- Chapter 1 - Object Theory
  [*Abadi96*] [*Blair91c*] [*Blair91b*] [*Blair91a*] [*Flanagan96*] [*Gallagher91*] [*Kent89*] [*Kent91*] [*Koenig95c*] [*Liskov81*] [*Liskov87*] [*Palsberg94*] [*Pratt84*] [*Rist95*] [*Snyder87*] [*Stroustrup93*] [*Stroustrup97*] [*Wegner87*]
- Chapter 2 - The Existing C++ Model
  [*Dijkstra68*] [*Flanagan96*] [*Joyner96*] [*Kesteloot96*] [*Martin93*] [*Palsberg94*] [*Stroustrup93*] [*Stroustrup95*]
- Chapter 4 - Dynamic Encapsulation Model
  [*Burdick89*] [*Burdick90*] [*Chen96*] [*Kent94*] [*Riggs89*] [*Snyder87*]
- Chapter 5 - The DC++ Programming Language
  [*Oliver93*]
- Chapter 6 - DECO
  [*Koenig95a*] [*Koenig96*] [*Palsberg94*] [*Plauger95*]
- Chapter 7 - Implementation
  [*Pratt84*] [*Wild97*]

- Chapter 9 - Conclusions
  [*Joyner96*]

- Appendix A - DC++ Grammar Summary
  [*Koenig96*] [*Stroustrup93*] [*Stroustrup97*]

Sources Grouped by Topic

These listings include both cited and uncited sources of information.

- Object and/or Type Theory
  [*Cook90*] [*Harris91*] [*Chen96*] [*Kent91*] [*Koenig94*] [*Partridge94*] [*Wieringa95*] [*Wegner92*] [*Agha87*] [*Hailpern87*] [*Snyder87*] [*Ossher87*] [*Wegner87*] [*Dahl87*] [*Goguen87*] [*Beech87*] [*Skarra87*] [*Abadi96*] [*Booch91*] [*Liskov87*] [*Elmasri94*] [*Baron80*] [*Palsberg94*] [*Barbier92*] [*Wolczko92*] [*Mariani91*] [*Blair91d*] [*Blair91c*] [*Gallagher91*] [*Blair91b*] [*Blair91a*] [*Ormsby91*]

- Dynamic Binding and Method Dispatch
  [*Johnson91*] [*Davis92*] [*Haddad95*] [*Huang92*] [*Voss94*] [*Lindholm97*]

- Distributed Object Techniques
  [*Chang95*] [*Daniels93*] [*Nascimento92*] [*Tanenbaum95*] [*Prabhakar95*] [*Clémençon96*] [*Tanenbaum92*] [*Fekete*] [*Mowbray95*] [*Hutchison91*] [*Orfali96*]

- Remote Procedure Calls
  [*Chang95*] [*Ford95*] [*Tanenbaum95*] [*Mowbray95*] [*Hutchison91*] [*Bloomer92*]

- Object-Oriented Languages and Environments
  [*Agesen95*] [*Ungar87*] [*Haddad95*] [*Pratt84*] [*Wilde91*] [*Kristensen87*] [*Madsen87*] [*Kahn87*] [*Liskov81*] [*Flanagan96*] [*Rist95*] [*Lindholm97*] [*Henderson94*] [*Madsen91*] [*Gallagher91*] [*Cook91*] [*Harland91*]

- C++ and Language Enhancements
  [*Johnson91*] [*Davis92*] [*Driesen96*] [*Grossman93*] [*Haddad95*] [*Koenig95c*] [*Koenig95c*] [*Koenig95b*] [*Stroustrup93*] [*Stroustrup97*] [*Waldo93a*] [*Martin93*] [*Cargill93*] [*Waldo93b*] [*Granston93*] [*Stroustrup95*] [*Koenig95a*] [*Koenig96*] [*Cook91*]

- Criticisms of C++
  [*Cook90*] [*Harris91*] [*Chen96*] [*Partridge94*] [*Martin93*] [*Cargill93*] [*Stroustrup95*] [*Voss94*] [*Rist95*] [*Joyner96*]

- Suggested Reading
  [*Aho88*] [*Beck90*] [*Silberschatz90*] [*Waldo93a*] [*Stroustrup95*] [*Tanenbaum95*] [*Wirth95*] [*Kent78*] [*Comer87*] [*Tanenbaum89*] [*Levine92*] [*McKusick96*]

- Documents Used for Writing/Formatting this Thesis
  [*Lesk79a*] [*Ossana79*] [*Kernighan79*] [*Lesk79b*]

Index

Although this index is full of strange quirks due to its being quickly auto-generated by sed, sort, and uniq, it should still serve as a useful quick reference tool. Numbers printed in **bold** show which page(s) a term is defined on.

About This Thesis

The development of the ideas and software for this thesis began in March 1996, and continued, in scattered bits and pieces of time, until December 1999. The written portion was completed in January, and edited, printed, and bound in February 2000.

The text of this thesis was written with vi and the diagrams were drawn with xfig. It was formatted using a customized combination of the GNU implementations of sed, awk, eqn, tbl, and troff, with the ms and mps macro packages, as well as a customized thesis macro package designed, implemented, and tested while writing this paper. The slides used for the defense of this thesis were prepared with the same tools and a custom set of slide macros.

The final printing of this thesis was hand-duplexed on a non-duplex laser printer and printed on 100% recycled (50% post-consumer content) paper in an attempt to save trees.

The majority of this document was set in a standard Times Roman font with a point size of 11 and a vertical spacing of 13 points.

This thesis may be obtained electronically in *PostScript* and other forms from `http://www.csh.rit.edu/~shaggy/thesis.html`. The source code to all software referenced herein may be obtained from `http://www.csh.rit.edu/~shaggy/software.html` or `ftp://ftp.csh.rit.edu/pub/members/shaggy/thesis`. Since the work of this thesis is an ongoing project, it is possible that changes may be made to it after this printing. Therefore it is advisable to check the web-site for updated versions of both the written text of this thesis and the associated software.

The author may be contacted with questions or comments about this thesis or the associated work via email at `shaggy@csh.rit.edu`.