Barbados: an Integrated Persistent Programming Environment

Tim B. Cooper

"But to this day I thank my friends for constantly reminding me of a very important word called Persistence. Persistence is to the quality of the character of man, what carbon is to steel." *- Billy Ray Cyrus*

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

October 1996

Basser Department of Computer Science
University of Sydney

Abstract

The aim of this thesis is to explain the concept of an Integrated Persistent Programming Environment (IPPE), why it is useful, and to describe the author's experience in building one. The concept of an IPPE is quite a new one, and the IPPE described in this thesis is quite different from other existing IPPE's.

An integrated programming environment (or integrated development environment (IDE)) is an application a programmer uses to automate various aspects of programming. It typically consists of an editor, compiler, debugger, build tool and other components.

A persistent system is one in which there is no concept of a file - just a huge memory shared by all applications in which objects and data-structures exist in their structured format (i.e. objects linked with pointers).

An Integrated Persistent Programming Environment is an integrated programming environment on top of a persistent system, such that source-code, executable code, compilers and data all co-exist inside a persistent store. An IPPE is both an application of persistence as well as a programming interface to persistence.

An IPPE is radically different from existing programming environments. There are many benefits which arise from it, including incremental compilation, interactive compilation, data-structure browsers, the provision of persistence, a single language for shell-level and high-level programming and others.

The author has implemented an IPPE, called 'Barbados'.

Acknowledgments

Many thanks to my supervisor, Dr Michael Wise for his support and encouragement, Mick Hollins for many informative discussions and the anonymous reviewers of various papers.

Thanks also to Jeff Kingston and the Basser Dept of Computer Science for helping me out after my aborted attempt to emigrate to Finland.

Thanks also to Tom Jones and Franc Carter, and Renata, for many stimulating conversations.

1. Contents

2. Introduction

This thesis will explain the concept of an Integrated Persistent Programming Environment (IPPE), why it is useful, and describe the author's experiences in building one.

2.1 Persistence

Definition *"Data-Structure":* A network of objects linked by pointers.

Definition *"Persistent System":* A persistent system is one in which the life span of data is independent of the life span of the process which created it. This means that the user perceives a single memory which persists (i.e. exists intact) on disk after applications close and even after the system is turned off. In this memory, data-structures do not require any special attention from the programmer (e.g. subroutines to flatten them to a file) in order to persist.

A persistent system is essentially a giant virtual memory, spanning an entire file-system, where issues of data clustering and data protection have been dealt with.

The motivation for providing persistence is to (a) relieve the programmer from the burden of writing code to convert data-structures from memory to disk and back again, and (b) to encourage applications to share data in this structured format rather than using files.

2.2 Integrated Development Environments

An integrated development environment (IDE) is an application a programmer uses to automate various aspects of programming. It typically consists of an editor, compiler, debugger, build tool and other components.

Without an IDE, the editor, compiler, build tool and debugger are all separate applications which the programmer needs to invoke explicitly (e.g. from an operating system shell). An example of a non-integrated programming environment is the UNIX programming environment.

The advantages of an IDE are that the computer can automate some of the sub-tasks of programming. For example, dependencies between program modules can be generated automatically and the debugger can link directly into the editor rather than relating to source-code via line-numbers.

2.3 Integrated Persistent Programming Environments

Definition "Integrated Persistent Programming Environment (IPPE)": An IPPE is an integrated programming environment on top of a persistent system, such that source-code, executable code, compilers and data all co-exist inside the persistent store.

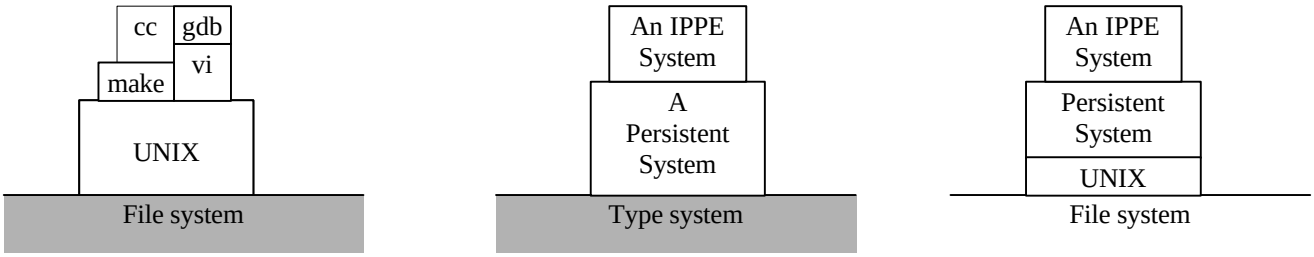
An IPPE is both an application of persistence as well as a programming interface to the persistent system.

Persistence has a lot to offer IDEs. The data-structures an IDE uses, for example executable code fragments, source-code fragments and dependency graphs would be difficult to manage without a persistent system. Persistence in fact makes it easy to manipulate program components at a finer grain, e.g. on a function-by-function basis, which assists in providing incremental compilation.

Conversely, I would argue that the full power of a persistent system is not realised without an IPPE. To program for a persistent system from outside that persistent system introduces a barrier between programs and data. This barrier can mean that extra checking of data types/formats is required in programs, non-source program components (e.g. GUI resources) cannot be as easily integrated into a program, debugging is more complicated, and so on.

An IPPE is radically different from traditional programming environments. The differences arise because an IPPE is capable of managing data at a finer grain than traditional environments. The system understands individual fine-grained objects (e.g. functions) and the relationships between them. Hence it can provide more services and provide services more efficiently. Some of the more obvious benefits which arise from an IPPE include incremental and interactive compilation, data-structure browsers, the provision of persistence, and a single language for shell-level and high-level programming. Other ways an IPPE can improve programming can only be understood from within the IPPE paradigm. Chapter 15 gives a fuller list of the benefits of an IPPE.

Comparing IPPE's to Traditional Systems



Traditional System

An IPPE

An IPPE compatible with traditional operating systems

2.4 Barbados

The author has constructed an IPPE, called the Barbados IPPE. Barbados will be referred to often in this thesis. The Barbados IPPE has been implemented on top of a custom-made persistent system (called the Barbados Persistent System). The whole system has been implemented in C, on top of the Windows 95/NT operating system.

2.5 Contributions of this Thesis

This thesis explores the concept of an IPPE. It describes what an IPPE is, what advantages and opportunities arise from it and what issues arise in its construction. It describes the Barbados system and how it differs from the only other IPPE known to the author, namely Napier.

Literature Review

3. Literature Review: Programming Environments

This chapter summarises the existing research and commercial programming environments apart from IPPE's.

3.1 Non-integrated Programming Environments

The UNIX programming environment is a good example of a popular, non-integrated programming environment [Ker84]. UNIX has influenced the design of Barbados in many ways: Barbados path-names are inspired by UNIX, the Barbados philosophy of 'many little tools' was adopted from UNIX, and the fact that the environment is based on a command-log is also inspired by UNIX. In UNIX, the various 'little tools' communicate mostly via text, whereas in Barbados the tools communicate directly using the type-system with structured data.

3.2 Integrated Interpretation Environments

With interpreted languages, it is easy to provide an interactive environment and a quick turnaround time. The most famous example of an integrated programming environment for an interpreter is Smalltalk.

The Smalltalk environment consists of a (mostly interpreted) object-oriented language also called Smalltalk, and a set of tightly integrated tools. Smalltalk is touted as having advantages over traditional systems because it supports interactive and incremental programming. It is also a completely object-oriented environment, which some would argue is a benefit. It is a mature and well-supported product, with a rich class library. It is claimed that it is effective for programming-in-the-large, because of its various features. Programs are supposedly developed by 'extending' the standard class library.

The Smalltalk environment is a Graphical User Interface (GUI) environment. The programmer typically has one or more 'browsers' open on the workspace, where a 'browser' is a window allowing the user to view and edit definitions of classes and class methods.

Wherever possible, the Smalltalk environment is 'modeless': this means that the programmer has all tools available at all times, as opposed to systems where the user enters and exits 'edit' mode, 'compile' mode or 'debugger' mode. This paradigm extends down to the language definition: the language is designed to support multi-threaded applications. Objects 'send messages' to each other, rather than calling member functions on each other. The difference is that the caller may or may not wait for the message to be processed, and in fact often does not wait.

The Smalltalk language is an extremely small language. Classes are objects, just as class instances are. To create an object one sends a 'new' message to the class object and receive a reply. Variables store either fundamental datums (integer, float etc) or act as references to larger objects. A variable is said to be 'bound' to an object, rather than 'storing' a value.

Smalltalk is dynamically typed and type-safe. Smalltalk code is mostly interpreted, meaning that there is a parsing stage which processes source-text into an intermediate form, but the majority of translation (e.g. method dispatch) is done at run-time. This means that it is slower than compiled environments and that more errors are reported at run-time than compile-time.

3.3 Integrated Compilation Environments

A large number of commercial 'Integrated Development Environments' (IDE's) exist for compiled languages. Modern examples include Microsoft Visual C++, Borland C++, Delphi and Symantec C++. Use of such IDE's is quite widespread.

An IDE consists of a single user-interface which integrates an editor, compiler and debugger. This integration brings various benefits, such as (a) automating program consistency (makefiles) and (b) tightly linking the debugger with the editor so e.g. the mapping between code position and source-code line is made visible to the user.

IDE's are also starting to provide more features, such as support for class-browsing tools and tools to construct GUI interfaces.

Barbados itself was constructed using IDE's of this form.

3.4 Hybrid Environments

One of the contributions of Barbados is to combine the best of both worlds of compilers and interpreters. The advantages of Barbados over interpreted environments are (a) compiled code is more efficient, and (b) many errors are detected before the program executes. The advantages of Barbados over compiled environments are (a) a quick edit-run-debug cycle and (b) being able to interactively execute commands which interact with programs.

Some hybrid environments exist, where there is both a compiler and an interpreter. This is partly the case with Oberon [Wir92] for example, and traditional environments where the debugger can execute commands in the language being debugged. Such environments provide some of the benefits of Barbados, for example testbedding of modules without compilation delays. However, they usually do not provide all the features of the language (e.g. macros) or execution of arbitrary code. It is also difficult to maintain both the compiler and interpreter and to keep them consistent with each other.

3.5 Incremental Programming Environments

3.5.1 Visual C++

The Microsoft Visual C++ 'integrated development environment' (IDE) is an example of the state-of-the-art in integrated programming environments [Mic96].

Barbados itself was developed mostly using the Borland C++ IDE and then using Visual C++. These two commercial IDE's are quite similar to each other: the user has

one ‘project’ open at a time, and the environment integrates the editor, compiler and debugger. Dependencies and makefiles are managed automatically.

Visual C++ has 4 technologies for achieving incremental compilation: (a) ‘incremental compilation’, (b) ‘minimum rebuild’, (c) ‘precompiled headers’ and (d) incremental linkage. Barbados provides the effect of all 4 mechanisms with one mechanism. I have found Visual C++ to compare poorly with Barbados in terms of incremental compilation efficiency. I also found a bug in the system when all 4 features were turned on. These issues are discussed further in the chapter on ‘Fine-Grained Builds’.

Visual C++ is not an interactive environment. Only whole programs can be executed, (i.e. beginning at ‘main()’). Code cannot be executed from the debugger. Only a limited form of expression evaluation is available from within the debugger.

3.6 Syntax-Tree based Incremental Programming Environments

All the research projects on incremental compilation/development environments which I am aware of, are based on the idea of using syntax trees to represent programs.

A syntax-tree is a data-structure which represents a parsed program as a tree, with nodes corresponding to grammatical productions. If a text version of a program is ever displayed, this is merely a ‘view’ on the syntax-tree representation. This view can aid incremental compilation of some sort, because it means that as programs are edited, it is not necessary to re-parse the unmodified parts of programs, and it is in some ways a more ‘natural’ representation of programs.

Examples of such systems include: IPE [Med81], Orm [Mag90] [Gus89], Gandalf [Not85], The Synthesizer Generator [Rep85] [Tei81], PSG [Bah86], Centaur [Bor88], Mentor [Don80], and Pathcal [Wil84].

I chose not to structure my environment in the same way, because (a) I was able to implement incremental compilation and other features quite easily without using syntax trees, (b) storing source-code as text is in many ways simpler than syntax-trees, and (c) using syntax trees can make it difficult to implement complex languages such as C++: it can be difficult to efficiently store inside syntax trees the amount of contextual information needed by such systems.

3.6.1 LOIPE (Gandalf)

LOIPE was an early incremental programming environment (hence the acronym) developed at Carnegie-Mellon university. It was associated with the Gandalf project [Not85].

LOIPE was an incremental program composition/compilation environment based on the syntax-tree representation of programs. The editor directly manipulated syntax-tree representations of programs. Program translation and code generation was performed as parts of the program were modified. The language supported was a variant of C.

3.6.2 PSG

The PSG system was an incremental programming environment which could work with many languages. Languages were described in a formal language definition,

(similarly to attribute grammars) and then environments for these languages were generated. The environment then consisted of a syntax-aware editor and a translator to incrementally translate programs into a functional language (which was then interpreted).

The editors could operate either as structure editors, in which syntax-trees for programs were directly manipulated, or as text editors in which syntax analysis was performed simply in order to detect errors early. The syntax-tree mode had the effect of preventing errors, while the text editor mode had the effect of immediately detecting errors (without forcing the user to fix them).

Incorrect or incomplete programs could still be executed. If execution ever reached an error or an incomplete fragment, the user would be taken into the editor and prompted to fix the error or fill in the missing pieces.

3.6.3 MultiView

MultiView [Mar90] is a project at the Flinders university of South Australia to develop an integrated software development environment. The focus is on unifying all software development tools with a single representation of a program, namely a syntax tree, and then providing various views on programs. Programmers can view and manipulate programs in a textual view, a function-call view, a flow-chart view and a generated code view, among other views. MultiView focuses on some higher-level features of a programming environment, e.g. support for modelling data and algorithms.

[Mcc96] describes a system for incremental code generation within MultiView. This system is based on fine-grained incremental compilation, namely at the sub-expression level. It uses a greedy algorithm, meaning that as programs are edited inside the environment, the changes are immediately updated in the compiled code.

3.6.4 'A Programming System'

[Buh86] describes a programming environment without a name but which has many features similar to Barbados. This environment supports a language which has features for manipulating files directly (which leads into persistence), it supports interactive programming using compilation techniques, and it manages program consistency transparently.

3.6.5 Orm

Orm is a sub-project of the Mjolner project [Dah87]. The aim of the Mjolner project is to 'increase the efficiency of the software development process in general' [Mag90]. The Orm project focuses on building an environment in which programs are developed incrementally. Other Mjolner projects involve programming language research (e.g. with the BETA language), software engineering methodologies and specification languages.

Orm is a graphical user interface (GUI) environment which supports incremental program construction. Programs are stored solely as syntax trees. They are edited directly in this structured form, using the GUI tools. For example, to write a loop, one clicks on the relevant icon, which brings up a template for a 'loop' production. This template includes several place-holders which the user fills in with other statements and expressions.

A lot of effort in Orm has gone into supporting the development of language grammars. Orm can be used either to develop programs in some programming language, or to develop language grammars in a formal grammar-specification language. The formal grammars include notations for specifying type and semantic information and code generation information.

Orm supports programming languages by interpreting the language grammars. It contains language grammars for Simula, parts of the BETA language, and other languages.

3.6.6 Centaur

Centaur [Bor88] is a system developed at INRIA, France. It is a system which generates programming environments from formal descriptions of languages. It uses the 'natural semantics' method of formally specifying languages. It has been released to over 100 organisations, mostly educational & research organisations, for experimenting with language design. Part of the focus of the project has been on providing support for formal theorem-proving notations. A syntax-checker for the C language has been implemented within Centaur and it is anticipated that a full Eiffel environment will be generated from Centaur at some stage.

As in the above systems, Centaur is based on a syntax-tree representation of programs. The generated environments are based around the syntax-trees and 'tools'. A tool is an object which manipulates a syntax-tree, such as an editor or pretty-printer or compiler.

4. Literature Review: Persistent Systems

This section gives a brief overview of persistence and persistent systems.

An object is said to 'persist' if it can outlive the process which created it. In traditional systems, objects created in RAM do not persist beyond the point where the creating process terminates. This is because in traditional systems, the memory used by a process is reclaimed when the process finishes. The only data which persists is data stored in files.

This means that if a program creates various data-structures in RAM, and it needs to store the data for a long period of time or to share it with another process, the programmer must have written code to store the data-structures in files. This process typically involves 'flattening' the data-structure into a linear sequence of bytes. It can be a tedious and error-prone task.

The field of persistence explores how this barrier between RAM and disk can be removed. A persistent system provides the user with a single memory, called a 'persistent store', in which data-structures can be created and manipulated and in which all data automatically persists until destroyed.

This concept may at first sound similar to virtual memory systems. However, virtual memory systems are not designed to encompass an entire file-system, as persistent systems are. If a virtual memory system was scaled up to encompass an entire file-system, with data being manipulated inside as if the memory were RAM memory, users would suffer problems of performance (because virtual memory requires locality of reference in order not to 'thrash' i.e. spend all processing time performing i/o) and problems of memory protection (because a single bugged program could potentially corrupt all data in the entire system).

So a persistent system is a single large, shared memory in which these problems of clustering data and memory protection have been adequately solved.

Persistent systems are usually closely integrated with compilers for high-level languages. Because a persistent system deals with data-structures, and data-structures are concepts related to types and high-level languages, persistence is a concept inherently related to programming languages.

When designing a persistent system, the designers must decide how fine-grained objects are to be stored, how pointers are dereferenced, and at what points data is brought from disk into RAM or migrated out to disk again. They must decide how to protect data from bugged programs and how to protect users from malicious programs.

4.1 Issues in Persistent Systems

4.1.1 Reachability (Garbage collection)

Because memory and disk space is never infinite, it is always necessary for a persistent system to have some system for re-using previously allocated space. In other

words, it must be possible to delete objects and reclaim the space they used for new objects.

The two main ways of doing this are (a) explicitly deleting objects, and (b) 'reachability': deleting objects once it is proved that they are not reachable via any sequence of pointer dereferences from a special object called a 'persistent root' (in other words they are not needed anymore).

If objects are to be explicitly deleted, then there must be some function call or program which implements the deletion/reclamation process. The C function 'free()' and the UNIX program 'rm' are examples of such functions. With this system, the programmer has the burden of determining when objects are or are not needed anymore, and if they make a mistake in this respect, the consequences can be bad: with hard-to-fix 'memory leak' or 'dangling pointer' errors.

If objects are to be deleted using the criteria of reachability, then the programmer does not have this burden. However, it means that a costly 'garbage collection' process must be run every so often, to detect unreachable objects. (There is no known efficient incremental algorithm for detecting unreachable objects). Also, it places restrictions on the languages and systems which are used, because it becomes necessary for the computer to be able to determine automatically and exactly the types and locations of all pointers.

There are solutions and workarounds to the disadvantages of both systems. This is discussed in chapter 13.

4.1.2 Hardware vs Operating System vs Library

A persistent system can be implemented at different levels: as a new hardware/software system, as an operating system, or as a library to be used in traditional operating systems.

A simple programming library for persistence can be called a 'persistent store'. A database which stores persistent objects and references between them is called an 'Object-Oriented Databases (OODB's)'. A 'Persistent System' in the strictest sense of the term is neither of these, but rather a programming environment integrated with an operating system (or runtime system which has aspects of an operating system), such that it provides the programmer with the abstraction of a single large RAM.

4.1.3 Orthogonality of persistence

The abstraction of a single large RAM can be provided at different levels of transparency. At one end of the spectrum, function calls to a database must be made. At the other end of the spectrum, issues relating to the interface between physical RAM and disk are dealt with inside individual pointer dereferences, and are completely invisible to the user. Related to this issue is the issue of whether persistence properties interact with the type system (e.g. do you have to inherit the property of 'persistence' from a special base class?) or if objects of any type are automatically able to persist. Also, there is the issue of whether you have to specify at object-creation-time whether the object is to be persistent or not.

4.1.4 Pointer swizzling vs other methods

One technique which is often used is ‘pointer swizzling’. Pointer swizzling refers to the action of modifying pointer values as objects migrate between address spaces. An alternative technique is to have a single very large address space, by having pointers of 64 bits or more.

4.2 TEXAS

TEXAS [Sin92] is a persistent store. It is implemented as a C++ library.

It implements persistence using a technique called ‘pointer swizzling at page-fault time’. This technique involves memory-resident pages containing pointers either to other memory-resident pages, or to pages marked as ‘not accessible’. If a pointer of the latter kind is dereferenced, the relevant page is read from disk into memory. When the page is read into memory, the system finds all pointers inside the page and ‘swizzles’ them. To ‘swizzle’ a pointer involves converting from a persistent format (pointing to the disk location of the other page) to a memory format. The memory format will either contain pointers to pages that are already memory-resident, or to pages of virtual memory which are marked as ‘not accessible’.

TEXAS provides transactions using this mechanism. TEXAS can also implement a ‘distributed shared memory’, that is a memory which spans more than one computer.

TEXAS can be programmed using traditional compilers, if they support run-time type information. This run-time type information is necessary for identifying pointers within pages.

4.3 ObjectStore

ObjectStore [Lam91] is a commercial object-oriented database. It is programmed either in ordinary C++, using library functions, or using ObjectStore’s own compiler which implements an extended version of C++.

ObjectStore implements persistence using techniques very similar to virtual memory. One or more databases can be mapped into a process’s address space, and accessed as ordinary data in virtual memory. ObjectStore uses the underlying system’s virtual memory hardware to provide its services. However it implements new virtual memory features:

- Concurrent access to database pages are managed, even across different machines.
- Transactions are supported using virtual memory.

In addition,

- ObjectStore provides various database features, such as managing bulk data types and relational database queries.

4.4 E and Exodus

E [Ric89] is a language which is a persistent version of C++. It was developed at the University of Wisconsin. E was designed as a database programming language.

E was built on top of a system called 'Exodus', which is a low-level object store. Exodus implements fine-grained access to objects. Specifically, there are function calls to Exodus to retrieve and store fine-grained objects in a persistent store. The E compiler then translates pointer dereferences in the source-code to calls to Exodus.

The E language has two parallel type-systems: one for transient data, and the other for persistent data. The persistent type system works like the ordinary C++ type-system, but with 'db' inserted onto the front of fundamental types and words such as 'struct'. The two type systems are provided so that ordinary transient data can be accessed with the full efficiency of C++ but persistent data can also be implemented.

The compiler then implements various optimisations to minimise calls to Exodus.

Barbados is similar to the E system in that they both have persistent languages based on C++. However, Barbados implements 'type-orthogonal persistence', meaning that objects of any type can persist. By contrast, in E, only objects created in the parallel 'db' type-system can persist. Also, Barbados implements persistence by using ordinary machine-code instructions for the majority of pointer dereferences, and having explicit syntax for those pointer dereferences which may involve disk i/o.

4.5 Shore

Shore [Car94] is a project which grew out of E and Exodus. Shore includes a persistent object repository which is similar to Exodus except that it understands the type system of the objects being stored.

One of the types in the type-system is the 'directory'. A Shore persistent store contains a directory hierarchy, similar to the UNIX directory hierarchy except with fine-grained typed objects inside the directories instead of files. The system maintains a strict hierarchy of directories, however it does support 'links' to directories and objects similar to UNIX 'symbolic links'. In these respects, it is similar to Barbados.

The unit of storage in Shore is an entity called a 'registered object'. A registered object consists of a contiguous sequence of bytes, which is organised into two parts: the 'object core' and the 'object heap'. The object core contains a single typed object, and the heap consists of additional typed objects which are reachable from the 'object core' but not from outside. The idea is that these additional objects are associated with the object core, and should be clustered with the object core to provide better performance. 'Registered objects' are given a system-wide 'persistent identifier', whereas the 'additional objects' cannot be referenced from outside the registered object.

The Shore system consists of a single type system, which can be accessed by multiple languages. If it is used from C++, then two types of pointers can be used: Ordinary C++ pointers which are used to denote additional objects, and 'refs' which are used to denote registered objects. 'Refs' are implemented using the C++ template mechanism.

4.6 SOS and Larchant

SOS [Sha89a][Sha89b] and its successor Larchant are 'Object-Oriented Operating Systems'. SOS supports a single type-system, but multiple languages can be used to access this type-system. The operating system manages objects instead of files, where objects are associated with a type and member functions. Objects, unlike files, can contain direct references to other objects.

SOS supports persistence, transactions and distribution.

4.7 Grasshopper

Grasshopper is an operating system being developed at the University of Sydney [Dea94]. It provides persistence by treating all data similarly to memory-mapped files. The basic persistent entity is the 'container'. A container is an address space which can be mapped into other address spaces. Data-structures can be built up inside a container, and then shared with other applications by being mapped into their address spaces.

There is no operating-system level concept of a 'type'. It is the applications which interpret the data in the containers. Each application can manage the data within a container in whatever way it sees fit: it can provide garbage collection, heap functions, database functions or other functions.

5. Literature Review: Persistent Programming Environments

There are a great number of persistent systems, of which some were described in the previous chapter. The majority of these are programmed with relatively standard programming environments, e.g. UNIX. Neither the source-code nor executable code is stored in the persistent store of these systems. Source-code is stored as ordinary files in the host operating system, and compiled usually with custom-made compiler. Such programming environments are of little relevance to this thesis, which deals with issues such as how persistence can aid program construction, incremental compilation and meta-level classes.

The first paper describing the concept of an IPPE was [Atk86], which was about the design of the Napier system focussing on language issues. However, [Mor94] is a more significant paper in terms of programming environments. Napier is the only other IPPE the authors are aware of, with the possible exception of Persistent Smalltalk (depending on the exact definition of an IPPE) and the PIPE project.

5.1 The Napier System

The 'Napier' system consists of a persistent system, programming language and programming environment.

5.1.1 The Napier Language

[Nap88] describes the Napier programming language. The PS-Algol language [Atk83] was to some extent the predecessor of the Napier language.

The Napier language was designed to be as simple as possible and orthogonal as possible. The idea was that every language mechanism should be able to be applied in every situation.

Definition "Type-Safe Language": A type-safe language is one which prevents the programmer from committing memory errors by providing strict typing rules. Memory errors include overwriting array bounds, corrupting the heap, and writing outside allocated areas of memory.

Definition "*Orthogonally Persistent System*": A persistent system in which all data is accessed in a uniform manner, regardless of its creator, longevity or type ([Atk83]).

The Napier language is type-safe. Type-safeness is used as the sole method of memory protection in the Napier persistent store.

The Napier language is also orthogonally persistent. This means that there is only one type of pointer, and only one granularity of object. This property considerably simplifies the language.

The type-system for the Napier language includes types for procedures and for entities called ‘environments’ which roughly correspond to Barbados directories. A Napier ‘environment’ is a collection of (name, type, value) bindings, but unlike directories as they are used in most systems, environments can be linked to form arbitrary graph structures, and there is no concept of a ‘hierarchy’ of environments or an ‘ownership tree’ as in Barbados.

Napier is not an object-based or an object-oriented language, since Napier ‘types’ do not include member functions. However, a user wishing to use object-based techniques can create types which include (as members) pointers to functions, and then initialise these pointers as each instance of the type is created.

A Napier identifier can be used within a code fragment to identify any object which is ‘in scope’. The objects which are ‘in scope’ at any time consist of local objects and objects brought into scope using the ‘use’ clause. The ‘use’ clause is a clause which brings named objects from specified ‘environments’ into scope for the following clause.

Napier contains many other features which are interesting from a language design point of view, however are not relevant to this thesis.

5.1.2 The Napier Hyperprogramming Environment

[Far92] and [Kir92] describe the Napier hyperprogramming programming environment.

This environment consists of a graphical user interface (GUI) ‘browser’ and ‘hyperprogramming facility’. The browser is a program which allows the programmer to interactively explore the persistent store, follow references between objects, examine values and execute code. The ‘hyperprogramming facility’ is a feature which allows the user to construct hyper-programs.

A hyper-program is a hyper-text document specifying a program. That is, it consists of a network of objects, where each object is a sequence of source-text containing embedded ‘links’ to other objects. These links are implemented with ordinary Napier references, but are displayed as hyper-text links. The links replace identifiers in the program, that is, they can be inserted anywhere where an identifier would be legal.

When a hyperprogram document is compiled, a ‘procedure’ is created which contains embedded links to the objects specified by the hypertext links and the identifiers.

The benefits of hyperprogramming include:

- Objects can sometimes be located more quickly using the mouse and the browser than by typing the access path to the object as text.
- It reduces the verbosity of Napier programs. In textual Napier, all objects used by a function must be explicitly brought into scope using a clause called a ‘USE’ clause. These clauses can be replaced by hyperprogram links.
- It leads to greater flexibility in program construction through a greater range of linking times; objects can be linked with a procedure during program execution, program linking, program compilation or even program composition.
- The sooner objects can be linked into a program, the sooner type-checks can be performed and errors detected.

5.1.3 The Original Napier Programming Environment

The Napier hyperprogramming environment is still in prototype form. The majority of Napier programming has been done in a more traditional way, outside the persistent store. Programs are written in UNIX as text files and then compiled and imported into the persistent store.

5.1.4 The Napier Persistent Store

[Bro92] describes the persistent store used by Napier.

The persistent store is implemented as a huge file on the host operating system. Data is read from disk in fixed-length pages. As pages are read in, pointers are swizzled from an 'on-disk' format to an 'in-memory' format.

5.2 Persistent Smalltalk

[Hos90a] and [Hos90b] describe the Persistent Smalltalk project at the University of Massachusetts. In this project, the Smalltalk run-time system was modified to support orthogonal persistence. Objects are clustered into units called 'physical segments' which are stored in the persistent store. The Smalltalk compiler was not modified. Programming environment issues were not explored in this project.

5.3 PIPE

PIPE (Persistent Integrated Programming Environments) was a short-lived project commenced at Adelaide University. It was closely related to the Napier project. It has evolved into a new project called 'Advanced Software Engineering Environments' under Dr Fred Brown and Dr Michael Oudshoorn. The focus of this new project is on the generation of programming environments from formal language specifications. The difference from other similar projects (Orm, Centaur etc) is that it uses persistence to support the system.

Description of Barbados

6. Description: The Barbados Programming Environment

In a nutshell, a Barbados system consists of a directory hierarchy containing (instead of files) ordinary C++ objects, e.g. ints and structs and classes and functions and so on. Their names follow the usual C++ naming rules for identifiers. Data-structures can hang off these objects, and they are stored on disk automatically when the user leaves the system, regardless of their type or structure.

There are no files, no "#include" header files, no modules, and no "makefiles".

The programming environment consists of a command-line, at which the user types C++ statements and declarations. To write a program, one simply writes a set of C++ classes or functions at this command-line (which doubles as a full-screen editor). To run a program, the user simply calls a top-level function.

6.1 Sample Session

This section describes the user's view of Barbados by providing a sample session with the Barbados system.

The interface to the Barbados IPPE is a text-based interface - essentially a command-line interface. Text in white represents the user's input; and text in grey represents the response of the system to the user's commands.

Simple expressions can be evaluated, as long as they are valid C++ expressions complete with the terminating ';'. If the expression returns a non-void value, this value is printed on the console:

```
3+4;
  = 7;                                                    // Output is in grey
```

In fact, any valid C++ statement can be entered. It is not compiled or executed until it is syntactically complete:

```
for (int j=0; j < 5; j++)
     cout << j;
01234
```

Declarations can be 'executed' as well. The following example shows a function definition being entered. A function or variable declaration/definition causes the specified object to be inserted into the current directory (Barbados has a concept of a 'current directory').

```
int f(int n)
{
     return n*5;
}
  f : function returning int (int);
```

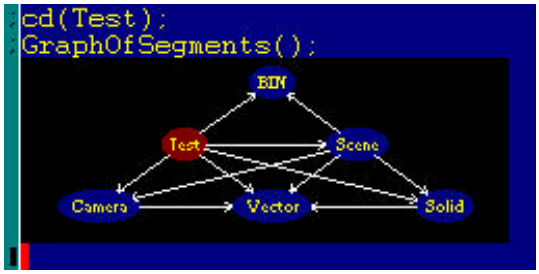
User-defined output methods can be invoked when values are automatically displayed. In the following example, we declare a complex number and then multiply it by itself. The appropriate 'operator*()' method is invoked, which returns a complex number. A print method has been written for the class 'complex', (either an 'ostream::operator<<()' method or a 'complex::Print()' method), and so this is invoked rather than the default method for printing structured values:

```
complex W(1,2);
  W : complex;

W*W;
  = -3+4i;
```

This feature is important because of the large impact that feedback has on software development. I believe that being able to clearly visualise complicated values has a large impact on the speed of the debugging process and on the speed of general interaction between human and computer. I also believe that it is a large benefit to see the output of a command displayed on the screen directly below the command, rather than in a separate 'output window' (where possible).

Barbados also allows bitmap output as well as text. These bitmaps are output to the scroll buffer, and scroll up along with text. For example:



6.2 The Command-Line

The command-line was designed so that users could use it both to enter short commands and also to enter long functions. The rationale for this was that for Barbados to be used in a highly interactive way, it would be cumbersome to have to swap between an editor and a command-line very frequently. Furthermore, a lot of functionality is shared between an editor and a fairly sophisticated command-line. For example, code for entering a single line and code to support entering multi-line commands is used by both.

An important part of supporting this kind of usage is the fact that the editor is capable of detecting when a syntactically complete, i.e. compileable entity has been entered. It does this using a superficial form of lexical analysis which is performed each time the user moves to a new line.

For example, the following code-fragments is a compileable entity:

```
A.Normalise();
```

as is:

```
int f(int n)
{
     return n * n;
}
```

Whereas the following is not:

```
for (int i=0; i < 10; i++)
```

When a compileable entity is entered, it sends the text comprising the entity to the compile/execute module. However, until this happens, the user is free to move up or down over the command or function-definition being entered and modify it.

Alternative ways of providing an interface to an interactive programming environment are:

(a) Implement a graphical user interface (GUI). This is the approach taken in the Napier and Self systems.
(b) Restrict all commands to one line. Many interpreters do this, (e.g. BASIC, FORTH), but obviously this is too restrictive for modern languages.
(c) Have the user specify when a command is complete. This could be accomplished with a special 'Compile Now' key, e.g. <keypad-ENTER>. This alternative would be appropriate if most commands were multi-line commands.
(d) Have the user specify when a command is *not* complete, for example adding a '\' to the end of each line if it is to be continued to the next line. This is the approach taken in many UNIX shells. It would be appropriate if most commands were single-line commands.
(e) Have a fully syntactically-aware editor.

Barbados detects when a compileable entity is complete by counting braces and searching for semi-colons. It is aware of the lexical structure of C++, e.g. it is not fooled by a brace inside a comment. It only examines each line as it is entered.

This feature can be fooled in some situations, e.g. by a macro including unbalanced braces. However, on the whole it has proven a very effective feature. Further details are provided in Appendix A.

Note that while C++ is amenable to this technique, other languages may not be.

6.3 Error Reporting

This kind of interface makes error-reporting easier for the compiler and easier for the user. As soon as an entity is completed, it is compiled, and as soon as it is compiled, any errors or warning are printed immediately to the console.

In fact, the first time the compiler detects an error, it communicates the position of the error to the editor. An error message is displayed just below the offending line, and the cursor is placed back on the source code in the precise position where the error occurred (or as close as the compiler can ascertain).

Suppose for example that an identifier is misspelled in a function definition. The user will be presented with an error message and the cursor will be put on the first character of the offending identifier. Small compilation errors (i.e. syntax and type errors) can be picked up very quickly and modified very quickly and with very few keystrokes.

A disadvantage of this is that the user cannot deliberately enter syntactically incorrect functions - they remain 'stuck' on the one piece of incorrect source code until it can be compiled. (At present, there is no way of bypassing a piece of incorrect source-code.)

6.4 History

A history of the most recent commands is kept. If the user moves the cursor past the top of the current compileable entity, the 'history' mode is entered. In this mode, previous compileable entities are displayed to the user as they use the 'up' and 'down' arrows. Any attempt to edit the entity will cause it to be inserted into the current position in the scroll-buffer.

6.5 Output of Values

Non-void expressions always produce output. Whenever an expression is entered and evaluated at the Barbados prompt, the system attempts to display it by calling a 'expr.Print()' method or an 'ostream << expr' method. If neither of these exist, it displays the value using a generic output function. It attempts to print values as far as possible in a format compatible with the language itself. For example, 'char's are displayed with single quotes and with escape sequences in accordance with C++ rules, as in the following:

```
(char)10;
 = '\n';
```

The "ostream<<()" operator is capable of automatically using 'Print()' methods, using a special rule implemented by the compiler. Therefore, in many cases the following two code fragments yield the same behaviour:

```
<expr>;
```

```
(void)(cout << <expr> );
```
[1]

Ordinary output from commands such as 'printf' are output to the same output stream.

The output of commands are displayed in a different colour to input (source text). It would be confusing if input and output were not visually discriminated.

Output lines also exist in the line buffer as ordinary text lines. The user can use the 'up' arrow to enter those lines. They are read-only lines, but it is possible to

[1] The '(void)' removes the expression's return value. Without it, the return value of the 'cout' expression would be displayed as well.

copy text out of them and paste into the source lines. This feature can be useful, especially since values are output in a similar format to the way they are denoted in the language.

Bitmaps can also be output, either by an explicit "cout <<" command or by the automatic value display mechanism. They scroll up in the scroll-buffer, along with lines of text. If the user writes a class for which the output would be too large to effectively fit in the scroll-buffer, they can construct a 'Print()' method to output a bitmap to a new window (as a side-effect) rather than displaying anything in the scroll-buffer.

There is a Barbados class called 'Bitmap' which can be used to manipulate bitmaps.

6.6 Input of Values

A common way for small Barbados programs to receive input is by function calls, rather than by functions such as 'scanf()'. This is because Barbados is an interactive environment and it is easier to receive input this way. For example, the user can look up a symbol table using the following call:

```
FindCustomer("Renata Kasasova");
```

This feature is especially convenient when code is being tested, because it minimises the framework necessary for testing.

The converse of a 'Print()' method would be a constructor method attached to a class which maps textual input to a value of that class. In fact, this can be provided from within the C++ language simply by writing a constructor which takes a single string (i.e. char*) as an argument.

For example, if the user wants to use the string "-3 + 4i" to represent a complex number, they can type the following:

```
W = complex("-3+4i");
```

In fact, the string: `complex("-3+4i")` can be inserted anywhere in a program where a complex value is required.

6.7 The Meta-Level Classes:

Four classes are defined to allow users to interact with objects in the Barbados system. These classes are designed to replace the C++ file-management functions, since in a persistent system typed objects are supposed to replace files.

These classes are also useful for extending the programming environment with new tools. The Barbados browser (described in Chapter 9) is an example of a program written within the Barbados environment based on the meta-level classes.

They are all ordinary classes, in the sense of being first-class objects within the language, however they were implemented at the system level because they require special privileges. They are:

type This represents the type of an object. A new keyword has been implemented, 'typeof()', which works similarly to 'sizeof()' to retrieve the run-time type of a language type-expression or instance. *type*'s can come from the typeof() operator, from the expansion of other types, or from other meta-level classes.	*any* This represents a (type, value) pair. It effectively means that the object is dynamically typed.
named_obj This represents a (name, type, value) tuple, or equivalently, a (name, any) pair. There are other fields contained inside a *named_obj*, for example 'storage class' and the 'make-info object', however these are mainly used internally.	*directory* This represents a directory. A directory is a collection of *named_obj*'s. These *named_obj*'s can themselves represent directories.

The operator '`typeof`' represents the link between the static world of C++ declarations / type expressions, and the dynamic world of '`type`' instances. As with the C++ '`sizeof()`' operator, both type expressions and values can be inserted into '`typeof`'. For example:

```
typeof(int);
 = int;
int i;
 i : int;
typeof(i);
 = int;
typeof(i + 8.0);
 = float;
```

An object of type 'any' represents an *l-value* to some object. ('l-value' is a C++ term denoting a reference to a modifiable object, such as can appear on the left-hand side of an assignment statement, as opposed to an 'r-value' which can only appear on the right-hand side of an assignment statement, for example integer constants and function return values).

In other words, if a value is assigned to an 'any', it must be an l-value. In this case, the 'any' takes on the type and contents of the value being assigned. (The class 'any' is implemented with just two data members: a pointer to the type and a pointer to the value).

This means that r-values, for example integer constants and return values of functions cannot be assigned to 'any's. This is a disadvantage, and it arises from the model of persistence used by Barbados: memory must be allocated and freed explicitly[2]. Therefore, to simplify the semantics of 'any' objects, it was decided to implement them as l-values.

[2] There is a form of garbage collection, described in Appendix B ('Persistence in Barbados'), but it is not a prominent part of the system.

Napier does not suffer this disadvantage. In Napier, an arbitrary r-value can be stored in an 'any' by creating a new object to store it. Barbados could use the same technique, however with Barbados's limited support for garbage collection, this technique could cause the system to run out of memory if used too frequently. Nevertheless, this alternative might still be considered in the future.

If an 'any' value is assigned to an 'any', it becomes a simple assignment of equivalent types (as opposed to an 'any' storing a value of type 'any').

The following fragment shows an example of the meta-level classes in action:

```
int num=9;
     num : int;
typeof(num);
 = int;
sizeof(num);
 = 4;
any A=num;
     A : any;
A;
 = { int, 9 };
A.Type() == typeof(int);
 = true;
if (A.Type() == typeof(int))
     num = A;
 = 9;
if (A.Type() == typeof(int))
     num = (int)A + 1;
 = 10;
```

A type can be expanded in the following way:

```
TypeExpander tyex;
     tyex : TypeExpander;
A.Type().Expand(tyex);
 = 'i';
tyex;
 = { 'i', NULL, 0, NULL, NULL };
```

The 'TypeExpander' is a class containing various fields for describing a type. A type can be expanded into a 'TypeExpander' object. A 'TypeExpander' expands a type expression one level. For example, a pointer type is separated into two parts: (a) an enumerated value denoting 'pointer' and (b) a 'Type' object representing the type being pointed to. In the above example, the type is a fundamental type namely '`int`', which is denoted by the value `'i'`. In the case of a struct or union or class, the 'TypeExpander' has another field which can be used to iterate over the members of the type. Each field of the type-definition is described with an instance of the class '`named_obj`'.

The presence of these meta-level classes makes it easy to extend the programming environment with tools which manipulate program entities and typed data.

Examples are given in Chapter 9. However, one notable example of such a program is the '`Browse()`' program. This program displays an arbitrary data-

structure as a graph in 2 dimensions, for use by programmers primarily during debugging.

'`Browse()`' was written using the meta-level classes, and a 2-D graph visualisation class, but without any special privileges. The entire program consists of 122 lines of code. In this code, type information is queried to separate each object into pointers and non-pointers. This information is then passed to the graph visualisation code.

Appendix C ("Meta-Level Classes' Interfaces") gives a full list of the interfaces to the classes.

6.8 The Callable Compiler

The Barbados compiler can be called from within the environment. It maps a string to a void function pointer. The return value can be executed, immediately or later, with the '()' C++ notation. However, for declarations and function definitions, the action of compiling the string is enough to cause the relevant object to be created.

For example:

```
compile("int f(int n) { return n*2; }");
 = 0x84f3f0;
compile("cout << f(9);")();
18
```

In the first example, the string is compiled, and since it is only a declaration, it does not need to be executed. The mere act of compiling causes the function 'f' to be inserted into the current directory.

In the second example, the extra pair of brackets is needed to actually execute the code after compiling.

A significant amount of research has been performed on exploiting callable compilers within the Napier project [Kir92b][Kir93]. The technique of constructing source-code representations of programs within a program, and then passing this text to the compiler, is called 'linguistic reflection'.

Applications of callable compilers / linguistic reflection include (a) implementing 'mini-languages' (application-specific languages), and (b) achieving a form of code re-use similar to but more powerful than ad-hoc polymorphism. Judging from the Napier experience, it seems that writing programs which utilise linguistic reflection can be quite a complex task.

The Napier Browser was originally implemented using a callable compiler, and later with meta-level classes.

6.9 Source Code

Source-code in Barbados is captured each time the user enters a compileable entity. If the compileable entity represents a declaration, the source-code is stored attached to the declared entity (or entities). If the compileable entity represents a statement, then the source-code is discarded.

Source-code for one declaration is stored as a single string in the Barbados persistent store. It is stored as a named object, of type 'char*' in a subdirectory called 'SRC' which hangs off whatever directory the declared object exists in. It has the same name as the corresponding declared object.

For example:

```
int f(int n)
{
     return n * n;
}
     f : function returning int (int);
dir();
/demo:
f()       SRC/
f(9);
 = 81;
SRC/f;
 = "int f(int n)\n{\n\treturn n * n;\n}\n";
cout << SRC/f;
int f(int n)
{
     return n * n;
}
 = cout;
```

In this example, a function 'f()' is defined. The act of compilation causes both the function f() to be inserted into the current directory and the string 'f' to be inserted into the 'SRC' subdirectory. The 'SRC' subdirectory is created in a particular directory the first time the system needs to store source-code for the directory.

6.10 The Standard Library Extensions

The following functions and objects constitute some of Barbados's extensions to the standard library:

`string`	A 'char*' typedef
`boolean`	An enum { false, true } typedef
`edit(string s)`	Retrieves the source for the named object back into the command-line
`compile(string s)`	Compiles the given string. All declarations/definitions are inserted into the current directory, and executable code is placed in a new memory block and returned. The return type is a function returning void and taking no parameters.
`cd(?)`	An overloaded function which is used to change the current directory. Can take a directory, directory, a string or other types.
`Help()`	Displays the operations available to the given object. It is a macro which uses 'typeof', so

	therefore it can be passed both a type or a value.
`Help2()`	Like 'Help()', but it also delves into source-code to retrieve the synopsis of the function in the form of comments.
`class Bitmap`	A class to help the user manipulate bitmaps.
`dir(?)`	An overloaded function used to display directory listings. Can take no parameters, (in which case it displays the current directory), or it can take a directory or directory LGO as a parameter.
`each_dir_entry (directory)`	A macro iterator (to be inserted into a 'for' loop) for iterating over members of a directory
`each_member(type)`	A macro iterator to iterate over members of a type-definition.
`sync()`	Saves all unsaved information to disk, i.e. commits all changes.
`GraphOfCalls(s)`	Displays a 2-D graph rendering of the direct and indirect function calls this function makes
`GraphOfDepends(s)`	Displays a 2-D graph rendering of the objects this object depends upon (directly or indirectly). It includes types, macros, global variables as well as functions.
`GraphOfLGOs()`	The following chapter describes large-grained objects (LGO's). This function displays a 2-D graph of all open large-grained objects, including their relationships to each other and whether each is a read or write copy.
`GraphOfCalls(s)`	Displays a 2-D graph rendering of the direct and indirect function calls this function makes.
`grepo(string s)`	Search for this string in the names of any object in the current directory or any child directories.
`greps(string s)`	Search for this string in the names or source-code of objects in the current directory or any child directories.
`grepdepends(s)`	Search for any object which depends on the given object.

7. Description: The Barbados/C++ Language

Although the Barbados persistent system was designed to support multiple languages, the Barbados IPPE is specific to C++. This chapter describes the Barbados C++ language and where it differs from standard C++.

7.1 C++ - why?

There are mixed feelings about C++ within the computer science world [Joy92]. Two of the more common complaints are:

- C++ is not type-safe. For instance, it has the notorious 'cast' operator. Array bounds are not checked. Memory bugs can lead to problems where the symptoms are far removed from the cause.
- C++ compromises object-orientedness in its attempt to straddle both structured programming (C) and object-orientedness.

The first complaint can be an advantage as well as a weakness. Being type-unsafe makes it very efficient: there are no run-time type checks or array index checking. It also makes it flexible, for example the user can write applications such as compilers and compression utilities which manipulate pointers in a type-unsafe way. Of course, in a persistent system, memory protection becomes very important: there is no longer the form of memory protection afforded by files. However, the Barbados persistent system has a concept of a 'large-grained object' which provides memory protection in a similar way to files. There are also other features which provide memory protection. These are discussed in Appendix B.

The second complaint may be partly true, however, it is arguably an advantage for a language to support both object-oriented and procedural programming.

I chose to implement C++ because (a) it is a popular language in its own right as well as being essentially a superset of the ubiquitous C language, and (b) it is a rich, object-oriented language. The Barbados project involves some language design, but the goal was not to design a new language.

If I had chosen not to implement C++, Java would be a good second alternative. Java was not chosen mainly because it was not available at the time.

7.2 Barbados's Extensions to C++

Barbados C++ has the following extensions to C++:

1. Support for Persistence, via three new operators and a new type-modifier keyword.
2. The type-system has been extended further by the addition of dynamic arrays (which grow during run-time).

3. Path names, which can be used everywhere an ordinary identifier can be used except in `goto` labels.
4. The Meta-level classes, e.g. `type, any, named_obj, directory.` (See the previous chapter).
5. New standard library functions/objects. (See the previous chapter).

7.2.1 The Barbados Type System

The Barbados type system has been extended in the following ways:

- In addition to ordinary pointers, there are LGO pointers. LGO pointers are involved in providing support for Persistence. They are described in section 8.2.2. To declare an LGO pointer to a type X, one types:
  ```
  X lgo* var;
  ```

- Arrays are no longer equivalent to constant pointer values. For example, the following can invoke different methods:

```
int A[10];                    int *A;
cout << A;                    cout << A;
```

- Dynamic arrays are supported. That is, the programmer can declare an array whose size is unknown at compile-time. They do this by writing a '?' instead of the array size. These arrays automatically re-size as read or write accesses are made to successively higher index values. (Dynamic arrays are implemented using the 'realloc()' function). For example:
  ```
  int A[?];
  ```

  There are no restrictions on combining dynamic arrays with other types. For example, one can type:
  ```
  int A[3][?][100];
  ```

  The motivation for providing dynamic arrays was to avoid having users implement dynamic arrays themselves with malloc() and realloc(). The problem with users doing this is that the persistence mechanisms would not know the type of these 'malloc()' blocks and therefore be unable to swizzle pointers. In short, dynamic arrays make C++ more type-safe.

7.2.2 Persistence Extensions:

Central to the model of persistence provided by Barbados is the concept of a large-grained object.

Definition *"Fine-grained object":* A 'fine-grained object' is a small, typed object such as an integer, class instance or function.

Definition *"Large-Grained Object (LGO)":* A 'Large-Grained Object' is a collection of fine-grained objects. The fine-grained objects must

each be reachable from a special fine-grained object called the 'root object' of the LGO. Large-grained objects can be thought of as having a type defined by the type of the root object.

This concept was introduced in order to (a) provide access control to data, e.g. permissions, locking and transactions; and (b) to provide an intermediate-level grouping of data for use by various applications and system tools. Large-grained objects replace 'files' in many ways.

The fact that Barbados has 'large-grained objects', and the semantics of 'large-grained objects' mean that Barbados does not provide orthogonal persistence. Some researchers would argue that this is a disadvantage. Discussion of this issue is given in Appendix B ('Persistence in Barbados'), and also in [Coo95] [Coo96c].

Along with the two granularities of objects, (fine-grained objects and large-grained objects), Barbados C++ has two types of pointers: ordinary pointers and LGO pointers. In this, it may appear that Barbados C++ is similar to the E programming language [Ric93]. However, there is a major difference between the approach each language takes toward persistence. In Barbados, the LGO pointers point to large-grained objects which contain data-structures linked with fine-grained pointers. In E, 'ordinary pointers' are only to be used between *transient* objects, and the 'persistent pointer' must be used between all *persistent* (fine-grained) objects. Therefore, Barbados provides type-orthogonal persistence, whereas E does not.

In Barbados, ordinary pointers are declared with a '*' in the declaration; and LGO pointers are declared with the word '`lgo*`' in place of the '*'. Ordinary pointers are dereferenced in the normal way, e.g. with the '*' operator or '->' operator or '[]' operator. LGO pointers are dereferenced with the 'OpenLGO()' operator, which needs a corresponding 'CloseLGO()' operator.

For example:

```
int *ip;
int lgo* is;

...

cout << *ip;

cout << OpenLGO(is,no);
CloseLGO(is);
```

The first "cout<<" line shows a normal pointer being dereferenced. The second "cout<<" shows a LGO pointer being dereferenced. The first parameter to OpenLGO() is the LGO pointer, the second is a boolean value denoting whether write access is required to the LGO. (NB :- Normally LGO's consist of a lot more data than a single integer, so this example is a little contrived).

It should be noted that it is not necessary to use OpenLGO() and CloseLGO() calls in order to use persistence in Barbados. A very effective way of avoiding them is to partition data into directories and use the 'cd()' command and path-names to move between directories. The 'cd()' command and the path-name

resolution commands contain calls to OpenLGO() and CloseLGO(), and they provide simpler semantics for simple tasks.

OpenLGO() can not be implemented as a simple function because of the way it manipulates types: the type of the return value is derived from the type of the first parameter. For example, in the above case the parameter is an LGO pointer to an 'int', and so the return value has type 'integer'. Therefore, OpenLGO() was implemented as an operator. The same applies to the operator 'CreateLGO()'.

If C++ templates had been implemented in Barbados, they would have provided an alternative implementation of OpenLGO() and CreateLGO().

When OpenLGO() is invoked, the LGO remains in memory until the corresponding call to 'CloseLGO()'. It is important to call 'CloseLGO()' when one has finished accessing an LGO because CloseLGO() performs the following tasks:

- It causes the LGO to be written to disk if write access was requested,
- The memory the LGO occupied is freed,
- The exclusive writer lock or shared reader lock on the LGO is revoked.

In this sense, OpenLGO() and CloseLGO() act as brackets around a transaction on a LGO. It is also the reason why pointer semantics are different for LGO pointers than for ordinary pointers. (Consideration was given to whether the '*' unary operator for dereferencing ordinary pointers could be overloaded to apply to LGOs pointers as well. However it was decided that it was necessary to have operators with 'open' and 'close' semantics).

CloseLGO() calls must match with OpenLGO() calls. For instance:

```
cout << OpenLGO(is,no);
cout << OpenLGO(is,no);
CloseLGO(is);            /* it is not yet closed */
CloseLGO(is);            /* Now it is closed. */
```

If the user attempts to dereference a pointer into a LGO after the LGO has been removed from memory, it is classified as a memory bug. Usually the invalid access will be trapped, (either by the virtual memory hardware or by the other forms of memory protection), and a run-time error will occur.

The operator 'CreateLGO()' is used to create new LGOs. Since all LGO's must be associated with a single root object, CreateLGO() creates both a fine-grained object and a large-grained object. The pointer to the fine-grained object is returned formally, and the pointer to the LGO is returned via a reference parameter. This parameter is also used to inform CreateLGO() of the type required.

The CreateLGO() operator is used in place of the 'new' operator. CreateLGO() requires two parameters: the LGO pointer which will receive the return value, and another LGO pointer corresponding to LGO which will be the parent of this LGO. Each LGO has a parent LGO, and these 'parent/child' links form a strict hierarchy.

The following code fragment shows CreateLGO() being used:

```
int *ip;
```

```
int lgo* IP;

...

ip = CreateLGO(/, IP);          // The root LGO (/) is to be the parent.
CloseLGO(IP);
```

The LGO remains in memory until a call to 'CloseLGO()'. In this sense, it works similarly to 'OpenLGO()'.

Each open LGO corresponds to a heap. When the programmer allocates memory, e.g. with a call to 'malloc()' or 'new', the memory must be allocated in a specific heap/LGO. The programmer can choose to either specify the LGO in the call to malloc() et al., or leave it unspecified in which case the memory is created in the 'default heap'.

```
x = malloc(IP, sizeof(int));        // Creates it in 'IP's heap.
x = malloc(sizeof(int));            // Uses the 'default heap'.
```

The default heap is a pointer to a heap corresponding to some open LGO. The default heap is set by one of the following functions:

OpenLGO()
CreateLGO()
SetDefaultHeap();

In Barbados, it is recommended that programmers use the default heap and the traditional syntax of 'malloc()', rather than the 2-parameter version of 'malloc()'. The reason is that (a) programs which simultaneously manipulate just a few LGO's will often be smaller this way, and (b) this syntax involves less modification to existing code.

For example:

```
class tree {
public:
     tree *right, *left;
};
     tree : tree;
tree  *root,  lgo* ROOT;
     root : pointer to tree;
     ROOT : lgo* of tree;
root = CreateLGO(/, ROOT);
 = 0x84f580;
ROOT;
 = B271;            /* An LGO pointer value */
root;
 = 0x84f528;        /* A normal pointer value */
root->left = new tree;
 = 0x84f60c;        /* Allocated in the new LGO */
root->right = malloc(sizeof(tree));
 = 0x84f62c;        /* Also allocated in the new LGO */
CloseLGO(root);     /* The default LGO now reverts  */
```

```
/* to the LGO containing the   */
/* current directory.          */
```

More details on the semantics of the persistence operators, for example how they work with sharing and transactions, are given in Appendix B ('Persistence in Barbados').

7.2.3 Scoping and Naming

The normal C++ rules for variables' scope, visibility and lifetime apply with local variables. (Local variables are any objects declared inside a function or compound statement).

In addition, any object in the current directory can be accessed with a simple identifier. These objects correspond to 'file scope' variables in standard C++. Note that the 'current directory' here refers to what was the current directory at the time the object was declared, which can also be thought of as the object's 'home' directory.

However, a simple identifier can be used to identify not just an object in the current directory, but also *any object inside the immediate child directories of the current directory.*

> **Definition***"name-space":* The name-space means the set of named objects which can be identified with a single identifier. (Not to be confused with the 'namespace' keyword in the draft C++ standard.)

The reasons for this naming scheme are:

To encourage layering of software:

Each function in each directory should ideally be constructed from general-purpose classes/functions, which are accessed using their full path names, plus more specific classes and functions which are implemented in subdirectories of the object's directory. The inclusion of child directory objects in the name-space encourages the use of objects in subdirectories, while the fact that this only applies to immediate children discourages users from 'jumping' a layer and directly accessing objects two levels away.

For importing modules:

In Barbados, a 'directory' corresponds to a module. If one directory needs objects provided by another directory, then the user can either denote them with path-names, or merely set up a reference from the first directory to the second (using a C++ 'directory&' object, e.g.: `directory &x = /...`). The latter alternative will make the second directory look like a child subdirectory, and so according to the above rule about immediate child directories, all the objects inside will immediately become part of the current name-space, i.e. to be used without qualification. This technique makes Barbados directories act like Ada 'packages'.

To avoid name-clashes:

The new 'namespace' construct in the draft C++ standard was designed to avoid name clashes, either compilation name clashes or loader name clashes. As programs and libraries become larger, name clashes become more common.

Barbados directories provide the same features as C++ 'namespace's, but I would argue it provides them more elegantly because it does not require modification to the language.

If there are multiple equivalent identifiers in the current name-space, then the identifier in the current directory always takes precedence over identifiers in child directories. If there are multiple objects with the same identifier in child directories, and this identifier is accessed without being qualified by a directory name, then an 'ambiguous identifier error' occurs.

In addition, the contents of the special directory '`/BIN`' are always in scope.

7.2.4 Path-names

Since Barbados is an integrated persistent system, and programs are constructed from fine-grained components that exist in a directory hierarchy, it was necessary to support path-names at a very low level. In fact, it should be possible to use a path-name wherever a normal C++ identifier is legal. A path-name should resolve to a named object, complete with type, storage class and location.

In Barbados, path-names operate in a very similar way to UNIX. The root directory in Barbados is '/' as it is in UNIX. Path-names can be either absolute, in which case they start at the root, or relative in which case they start with some object that happens to be a directory and happens to be in the current directory (or indeed, any object which can be denoted by a simple identifier).

A path-name consists of a series of directory names, separated by '/' symbols. A path is resolved by starting at the first directory, taking the next name, looking that name up in the directory, and continuing. All the identifiers in a path-name, except the last one, must be directories of some kind: either a normal Barbados directory, a reference to a directory, or an LGO pointer to a directory.

Relative path-names can also incorporate the '..' symbol, meaning the parent of the current (or denoted) directory. The '..' on its own can also be used to denote the parent of the current directory, and a '.' on its own can be used to denote the current directory (although it is always redundant in a path name).

For instance:

```
/cl/Vector V;
output/xl;
../barrel;
..;
.;
/;                          // The root directory
```

All the above are valid Barbados statements, since they resolve to named objects and therefore valid expressions. The values of those objects are printed on the console in each case.

The 'Print()' method for directories displays a directory listing. Therefore, the following two code fragments are equivalent:

```
dir();
.;
```

A directory listing displays the directory name, and generally about 4 columns of named-objects. A limited amount of type information is displayed for each named object, for example:

```
f()             // A function
SRC/            // A directory
n               // A variable
#max            // A macro
complex         // A type
```

When a path-name occurs in source-text, outside strings and comments and so on, it is resolved at compile-time. If no such object exists, an 'undeclared identifier' error will result. For example, the following will yield an error:

```
int f()
{
     D.Create("foo", typeof(int));        // Create:  D/foo
     D/foo = 4;
}
```

Therefore, path-names cannot be used in this way to denote objects which are created at run-time. Run-time evaluation of path-names is provided by the 'Find()' and 'FindAny()' member functions of the class 'directory'. For example, the following is legal:

```
int f()
{
     D.Create("foo", typeof(int));        // Create:  D/foo
     (int)D.FindAny("foo") = 4;           // Assign to it.
     // Or:  (int)(.).FindAny("D/foo") = 4;
}
```

If Barbados is being used as a shell language, e.g. for file management, the compile-time path-names are sufficient. This is because commands are being executed at the same time as they are compiled, and so the compile-time context is the same as the run-time context. For example, the user can type:

```
delete D/foo;
```

Path-names can resolve to any named object, including type-definition objects:

```
/cl/f(5,4);          /* A function object */
/cl/bar += 1;        /* A variable object */
/cl/Vector V;        /* A class object */
```

The lexical analyser removes any confusion between the '.' token and '.' as used in float constants. The parser then distinguishes between '.' when used to denote the current directory and when used for member access or ellipsis dots.

Note also that the '/' token is now overloaded. It is used both as the path-name separator and as the division operator. How are ambiguities like the following resolved?

```
a = /cl/b / 2;
a = b / c;
```

The compiler relies on type information to remove confusion. If an identifier is followed by a '/', and it is a directory, then it is interpreted as a path-name. Otherwise it is interpreted as an operator/(). Therefore it would be foolish to write an operator/() which operated on directories, because it would not be possible to use it except by enclosing the operands in brackets.

Path-names are implemented at a level in-between lexical analysis and parsing. Therefore the same mechanisms allow path-names to be used to denote types in declarations as well as objects in expressions. However, when an object is being created, it is not legal to use a path name. For example, the following is illegal:

```
int cl/f(int n)
{
     return n + 2;
}
```

..as is:

```
#define cl/min(a,b)  (a<b)?a:b
```

The reason for this restriction is that Barbados assumes that each object is compiled in the same directory it exists in. This directory is called the object's 'home directory', and is defined as the directory in which the object was first declared. It is important to use the same context (i.e. name-space) when recompiling objects. (Note that only named objects are compiled, and that each named object is guaranteed to belong to a particular directory. Dynamic objects of course are neither compiled nor belong to any particular directory).

Use of the '\' backslash character was considered instead of the forward slash for use in path-names. One advantage would be that it removes the ambiguity with the division operator. '/' was chosen instead because it is more prominent on most keyboards, it is the character used in UNIX which inspired these path-names, and backslash causes complications when path-names are represented as strings (as they will often have to be in Barbados), because of C++'s special interpretation of '\' in strings. However, there are advantages either way.

7.3 Barbados's intentional omissions from C++

Barbados C++ is not a strict superset of C++. Some features of C++ have been disabled, (though they will still be parsed), because they correspond to concepts which are no longer meaningful. Other features were not implemented simply because of lack of time.

The following features are omitted because they are no longer meaningful:

- ‘extern’ and ‘static’ keywords (except when ‘static’ is used within a function or class).
- #include
- namespaces

In the Barbados paradigm, directories are used rather than ‘modules’. If objects in one directory wish to access objects in another directory, then path-names or naming rules are used to retrieve the object.

As a result, there is no need for header files or ‘extern’ or ‘static’ storage classes (except when ‘static’ is used within a function or class). If one wishes to use a certain set of functions or objects, it suffices to create a link from the current directory to the directory where those objects reside. This link will pull them into the current name-space. This feature is very similar to the concept of ‘importing’ modules, e.g. in Ada. It saves the programmer from having to keep function definitions consistent with their prototypes.

There is no ‘persistent’ keyword or storage-type as in the E language. In Barbados, every object automatically persists until explicitly deleted (although note that deletion can occur both at the fine-grained level and at the LGO level). Note also that there is a special heap, called the ‘temporary heap’, where objects are understood to be transient.

7.4 Barbados’ temporary omissions from C++

There are other omissions Barbados C++ has made from standard C++. The following features are not supported yet, due to limited resources:

- Virtual functions
- Virtual inheritance
- Proper constructor/destructor semantics
- Templates
- Bit-fields
- in-line functions
- miscellaneous features, such as doubles, const/volatile, signed/unsigned.

8. Fine-grained Builds

(This chapter is to be published in Software: Practice and Experience as "Incremental Compilation through Fine-Grained Builds")

8.1 Introduction

Suppose a programmer writes a class definition, then writes a function which uses that class, and then modifies the class definition. The following code fragment provides an example of such a situation.

```
typedef struct {
     float x,y;
} vector;

float length(vector v)
{
     return sqrt(v.x*v.x + v.y*v.y);
}

typedef struct {
     boolean rel_or_abs;
     float x,y;
} vector;


vector v = { true,3,4 };

length(v);
```

Unless the function 'length()' is recompiled following the modification to 'vector', the function will be out of date when called and work erroneously.

> **Definition** *"A depends on B":* We say that object A depends on object B if a modification to B will cause A to need recompilation.

Fortunately, Barbados transparently tracks all dependencies between program components and schedules necessary recompilations, so that whenever the user calls a function, it is guaranteed to be up-to-date with all other objects and with its source-code.

The fine-grained build feature applies to the above example in the following way:

```
typedef struct {
     float x,y;
} vector;                    // 'vector' is compiled now.

float length(vector v)
{
     return sqrt(v.x*v.x + v.y*v.y);
```

```
}                              // 'length' is compiled now.

typedef struct {
     boolean rel_or_abs;
     float x,y;                // The old definition of 'vector' is
} vector;                      // replaced with this one.


vector v = { true,3,4 };       // This statement is compiled and
                               // executed.

                               // At this point, 'length()' is out-of-date.

length(v);                     // Barbados automatically detects
 = 5.0;                        // that 'length()' is out-of-date,
                               // so it recompiles it and executes the
                               // new version.
```

In traditional programming environments, the unit of compilation is the 'file' or 'module'. The dependencies between modules are stored in a file, typically called a 'makefile', and a program called a 'build tool' processes the dependencies and schedules necessary compilations to bring a program up-to-date.

However, in an IPPE, the program components are much finer-grained. This difference in granularity leads to very different requirements of a build-tool. For example, the user cannot be expected to invoke a 'make' tool on every command, nor can the user be expected to manually maintain dependency-lists (e.g. as UNIX programmers commonly do).

Therefore, an IPPE must either use an interpreter instead of compiler, or it *must* have a fine-grained build tool (it would be unusable otherwise). This chapter is about how Barbados provides this feature, and how this feature is used to achieve incremental compilation.

8.2 The Problem Statement

I propose that any fine-grained build tool should satisfy the following requirements:

(a) Generate dependencies automatically.
(b) Bring or keep all program components up-to-date transparently.
(c) Ensure that there are never multiple versions of a compiled object (i.e. products of the same source code), in the program development area.
(d) Ensure that no out-of-date code is ever executed.

8.3 Some Subtleties

Reasons why the implementation of a fine-grained build tool can be complicated, (and why algorithms used in traditional make tools e.g. UNIX 'make' cannot be applied) include:

(a) Dependencies must be generated automatically (they are too numerous for the programmer to generate them).
(b) Functions, type/class-definitions, variables and preprocessor macros are all involved in the process.
(c) An entity can change size after a recompilation, and so depending on the implementation, it might change address.
(d) The tool must deal with recursive types and recursive functions.
(e) The tool must deal with thousands of entities (or more) rather than tens of entities.
(f) Objects can be modified at any time, so whereas traditional compilers can rely on the static source code to order the compilation of objects such as structures[3], a fine-grained compiler cannot.
(g) When objects are deleted, this could potentially cause dangling pointers in dependency lists.
(h) Objects may even become out-of-date during the recompilation process, e.g. by a function parameter value changing, which has implications for the ordering of recompilations.
(i) Dependencies can change during a compilation.

8.4 My Solution: Generating Dependencies

The Barbados solution to the fine-grained build problem is presented in two halves: how Barbados generate dependencies, and how it processes them. The following section deals with dependency generation.

8.4.1 The entities that the build tool must deal with

In the C++ language, there are the following types of entities which the build tool must deal with. (Other statically typed, compiled languages will have similar entities).

`typedef struct { int x,y; } Point;`	*`/* A type or class */`*
Often depends on other types, because of pointer fields and nested objects. Function prototypes can depend on types/classes.	

`Point P;`	*`/* A static variable */`*

[3] e.g. to generate apropriate sizeof()'s and field offsets,

Only depends on the type; although other objects can depend on it. If you change its type, this can affect everything that depends on it.	

`Point Point::Reflect()` `{` `  x = -x;` `}`	*`/* A function */`*
Consists of a prototype and a body. If the prototype changes, e.g. you modify a parameter type or return type, this can cause substantial changes in other functions which use it. If the body changes, at most this can change the address (which can be dealt with in various ways).	

`#define min(a,b) \` `  ((a < b) ? a : b)`	*`/* A preprocessor macro */`*
Does not depend on anything. On the other hand, any object which uses a macro is said to depend *directly* both on it and the objects generated in its expansion - which can only be determined during the compilation of that other object.	

In the remainder of this chapter, all of these entities will be referred to collectively as ‘make-objects’ or simply as ‘objects’.

In my system, the member functions of classes are considered to be make-objects in their own right. This means that dependencies can be expressed specifically to the member functions rather than to the class definition. The alternative was to lump them together with the class, for dependency purposes, but this was considered too large-grained to be desirable - it would generate too many recompilations.

In C++, the above entities are precisely those entities which have source-code and can be separately compiled. Therefore, they are the entities used by the build tool. Local variables, even local static variables and ‘goto’ labels do not participate in the build tool. In my system, immutable system functions such as ‘strcpy()’ also do not participate in the build tool. The above taxonomy would also apply to most other modern compiled languages.

8.4.2 Dependencies

Object ‘A’ is said to *depends directly* on object ‘B’ if a change to object ‘A’ would cause object ‘B’ to become out-of-date. ‘Out-of-date’ means that ‘B’ will change (or might change) if it were recompiled.

Therefore, an object depends on any other object or type-definition that is directly used by the object. This includes every function called, every static variable accessed, every type mentioned. In an object-oriented language, it also means the methods used - for example:

```
X * Y;          /* might use: */ operator*(complex, complex)
```

The phrase ‘dependency list’ will refer to the set of objects which a given object directly depends upon. The dependency list for an object can be generated quite easily during its compilation.

8.4.3 The ‘Interface’ optimisation

In UNIX, an object’s time-stamp is updated whenever it is recompiled. This property is tantamount to asserting that the object changes each time it is recompiled.

An effective optimisation is to separate an object’s *interface* from its *body*. The *interface* would be defined as that part of the object capable of affecting other objects, (in the static world of compile-time). In practice, this usually means a function’s prototype. Depending on how functions are implemented and how recursion is dealt with, the address of a function might also belong in the interface, since this is part of the view that the outside world sees.

An object would only be said to change if its interface changes. Interfaces change relatively infrequently. Therefore, making this distinction dramatically reduces the frequency of objects being out-of-date and needing recompilation. (In this framework, it is desirable for *interface-change-stamps* to be as old as possible and *compile-stamps* to be as new as possible).

Depending on which build algorithm is used, this distinction can be a necessity rather than an optimisation because of recursive types and functions.

8.4.4 Being up-to-date

Each object is given a time-stamp which marks the last time its interface changed, as well as a time-stamp marking the last time it was compiled.

An object is out-of-date, i.e. requiring recompilation, *if it depends on an object whose interface has changed since this object was compiled*.

8.4.5 Generating dependencies

The process of generating dependencies comes down to the compiler marking all the functions, static variables, types and macros that it comes across during the compilation of an object. A dependency of the form ‘A depends on B’ can either be stored in a list associated with ‘A’ or a list associated with ‘B’. Because of the top-down nature of my algorithm, it was preferable to store dependencies with the object that depends, rather than with the object that is depended upon.

8.5 My Solution: Make()

In the remainder of this chapter, the Barbados build tool will be referred to as ‘Make()’. The application programmer will be referred to as the ‘user’, since they are the user of Make().

8.5.1 When is Make() called?

Make() is a completely transparent feature. This means that the user is never required to explicitly invoke it and the user is not informed that it is even operating.

Make() provides the abstraction of every object being always up-to-date and guarantees that no out-of-date code is ever executed. This means that the compiler resembles an interpreter. However, unlike interpreters, errors are reported early, namely whenever the user executes code whose transitive closure includes an object with an error.

Whenever the user enters a compileable entity at the command-line, that entity is compiled. Sometimes that entity also needs to be executed, e.g. expressions and other statements.

Make() operates *lazily*. This means that objects are only recompiled when they are needed. When Make() is called, it is called recursively over all the objects accessed in the transitive closure of the code the user wants to execute. This means that the user can make multiple modifications to various objects, and compilation is only scheduled when the user calls some affected code.

The dependency list for an object is updated each time it is compiled.

Make() is never called at run-time - it is purely a ‘compile-time’ or ‘development-time’ function. Admittedly, the run-time/compile-time distinction is quite blurred in an interactive system such as this one: what this distinction means is that once user-code begins execution, Make() is never invoked again until control returns to the Barbados command-line.

For example:

```
typedef struct {          // A type-definition.  The compiler is
     int x,y;             // called, but Make() is not, since no
} Point;                  // code is generated.

Point P;                  // A variable declaration.  The compiler
                          // is called, but Make() is not.

Point Point::Reflect()    // A function definition. The compiler
{                         // is called, but Make() is not, since no
     x = -x;              // code is generated for immediate
}                         // execution.

P.Reflect();              // An expression. The compiler is called,
                          // then Make(), and then the code itself
                          // is executed.
Point P=Q+C;              // A declaration that includes an
                          // expression.  Calls the compiler,
                          // Make() and then the code itself.
Point A,B,C;              // A multiple declaration. Only the
                          // compiler is called.
```

8.5.2 The 'root' object

The purpose of one invocation of Make() is to bring a given 'root' object up-to-date. This root object could be a single function, e.g. a function called '`main()`', or it could be a statement issued at the command-line in interactive mode, for example:

```
A() + B() + C();
```

In this case, this command is wrapped up in a function object called '`_top_level`' which has dependencies to all three functions. To bring '`_top_level`' up-to-date is to bring every object needed for the current command up-to-date.

Sometimes, an object will need to be recompiled twice: once to regenerate dependencies, and then again if Make() causes types (and hence methods) to change. This situation can also happen to '`_top_level`', which is why Barbados needs to capture source-code not just for declarations but also for statements.

8.5.3 My Algorithm:

A first approximation to a build algorithm could be: "recompile everything that is out-of-date".

This algorithm is not sufficient, because objects can become out-of-date during the recompilation of another object, even if they have been compiled once already during this call to the build tool. For example, suppose the type `complex` and the object `complex F(complex);` are both out-of-date. If the function is compiled first, the subsequent compilation of the type could cause the function to be out-of-date again.

This illustrates the fact that the order of compilations can be important. As a second approximation, therefore, suppose we take the root object and follow the transitive closure of its dependencies. When we reach an object with no dependencies, we rise up the dependency graph, recompiling every out-of-date object as we go.

This algorithm has the benefit of compiling an object's antecedents before compiling the object itself. However, this algorithm is insufficient for two reasons:

(a) The first reason is that it doesn't deal with circularities in dependency graphs (circularities arise through recursive functions or types). Complicated recursive types might mean that some objects actually need to be compiled twice or even more times[4], so it is not possible simply to ignore dependencies to objects already reached.

(b) The other reason is that a compilation of an object can actually cause that object's dependency list to change. For example in C++, a change in a macro or in a type can cause that function to *depend directly* on different objects. These objects newly-inserted into the dependency list will not have been checked.

[4] Actually - this is implementation-dependent: if all objects are referenced by pointer-pointers and pointers to classes are distinguished in the dependency lists from nested structs, this situation can be prevented.

Therefore, my algorithm detects circularities (by marking objects reached) and doesn't descend into objects already examined during the current run of Make(). It also repeats the whole process as many times as it takes before there are no more compilations. This algorithm solves both problems.

This algorithm may appear at first to be inefficient, and to cause unnecessarily many recompilations because recursive objects will be compiled more than once. The usual way of dealing with recursive functions and functions used before they are defined is to have a separate back-patching or a linking phase [Aho86], whereas here, successivecompilations of an object effectively insert the recursive references. However, this extra amount of recompilation will seldom amount to a significant delay in response, and changes do not propagate around a cycle of recursive references. Only recursive code requires more than one compilation during one run of Make(), and only very complex recursive code requires more than two compilations during one run of Make().

The typical scenario is thus: The transitive closure is performed for the first time, during which the majority of out-of-date objects are recompiled. During the second pass, there may be a few more recompilations necessary because a recursive type or function changed its interface during the last pass or because dependency lists changed and new objects were introduced. In the third pass, no recompilations will be needed or performed, and so there will be no fourth pass.

It is my experience that the time spent doing this graph traversal, even multiple times, is insignificant compared to the time spent on recompilations.

8.5.4 The 'Check-stamp' Optimisation

According to the above algorithm, each execution of a program would require a traversal of its transitive closure - every component object it uses. Although this can be quite quick even for very large programs, I have implemented one optimisation to avoid this traversal. Each object is stamped with a third time-stamp called the 'check-stamp'. This stamp indicates the last time it was examined by Make(). This stamp is mainly used to avoid cycles when Barbados looks at the transitive closure of an object.

Barbados also logs the last time there was any change to an object's interface in the whole system. If an object's check-stamp is more recent than this global 'interface-change' stamp, then it is impossible for that object to be out-of-date. Therefore Barbados can skip even the transitive-closure checking stage. This optimisation is handy for long streams of little commands being entered interactively.

8.5.5 Example

To provide an example of my algorithm in action, and also to discuss recursion, consider the following code:

```
struct VALUE {
     int a,b;
     struct TREE *parent;
};

struct TREE {
     struct TREE *left, *right;
```

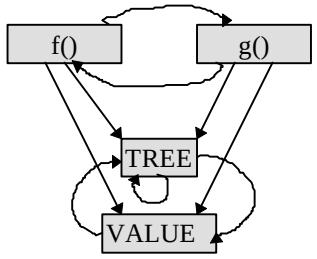
```
     struct VALUE val;
};

VALUE f(TREE t)
{
     if (t->left)
          return g(t->left);
     else return t->val;
}

VALUE g(TREE t)
{
     if (t->right)
          return f(t->right);
     else return t->val;
}

TREE t;
...
f(t);                    /* The build tool is first activated here. */
```

The dependency graph looks like this:



The algorithm performs as follows:

1. The build tool starts at ‘f()’.
2. Suppose it descends to ‘TREE’ first and then from ‘TREE’ into ‘VALUE’.
3. ‘VALUE’ will have dependencies back to ‘TREE’, however these are not followed since ‘TREE’ has already been reached on this pass.
4. Therefore the recursion bottoms out. ‘VALUE’s time-stamps are examined, and since ‘TREE’ was changed after ‘VALUE’ was last compiled, ‘VALUE’ is deemed out-of-date and it is recompiled. Since the size of the type-definition has not changed, it returns to the same location as before.
5. Returning up a level, ‘TREE’ is then examined. It is likewise out-of-date, and it is recompiled.
6. Similarly for ‘g()’ and finally for ‘f()’.
7. Since there were some compilations (which could have affected interfaces or dependency lists), the entire pass is repeated.
8. This time only ‘TREE’ is recompiled. This is because all the other objects are up-to-date, and yet ‘TREE’ depends on ‘VALUE’ and ‘VALUE’ had a recent

interface-change when the reference to the ‘TREE’ stub was replaced with the new ‘TREE’ object.

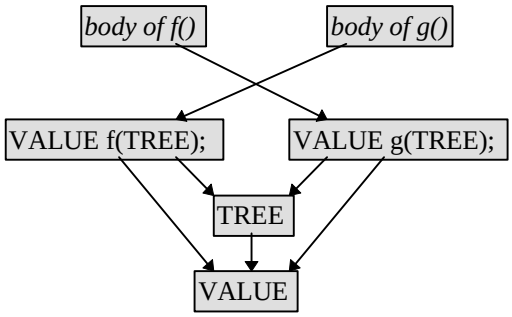
An alternative solution is described below, with reasons why I chose to adopt the above algorithm.

8.5.6 An alternative algorithm

Suppose we separate function bodies and function prototypes. Callers would point to a function header rather than a function body, and the body would be free to change size or implementation without affecting the prototype. A similar approach would then be used for types, for example a structure definition which includes a pointer to another struct would point to a ‘prototype’ for that struct which specifies the size of the function and a pointer to the actual structure definition. Dependencies would then be expressed between these objects (mostly from a body to a prototype). Furthermore, instead of a ‘compile stamp’ and an ‘interface-change stamp’, there is a single stamp on the ‘body’ object and a stamp on the ‘interface’ object.

Incidentally, this kind of approach has a similar effect as having multiple types of dependencies, e.g. Eiffel discriminates between client/supplier dependencies and inheritance dependencies.

For example, the above example would be rearranged as follows:



This structure has the advantage that any valid program would be represented by an acyclic directed graph (at least after the initial compilations during which dependencies are generated).

Furthermore, a build algorithm can be written which discovers dependencies on the fly and yet is guaranteed to terminate:

a) Recursively descend the program graph.
b) Upon ascent from a recurse, compile an object if it is out-of-date.
c) Compare dependency lists before and after a recompilation. Whenever they change, recurse again down the new dependencies and return to part (b).

This algorithm works because given an existing program graph, a change in dependencies can only affect objects higher in the dependency graph.

Each function call will then require a run-time indirection from the function-object pointer to the code itself, or a linking phase to make these conversions. The linking phase would be applied after all recompilations have been made, and it would be applied to all objects which were recompiled since the last linking phase.

Dangling ‘function call’ pointers from old functions to invalid addresses will never be invoked because the build tool would prevent it.

8.5.7 The choice of algorithm

Of these two algorithms, I chose to implement the first algorithm in my system because

(a) It is a little awkward to separately compile a function body and its prototype. (Similarly for structs, if a body/interface distinction is made).
(b) It was desired that CALL instructions would contain pointers to the physical location of a function’s body; whereas the above algorithm would require the use of pointer-pointers to function bodies (so that when an object increases in size and needs to be moved elsewhere, existing references remain valid).

Note that it was important to implement a ‘lazy’ algorithm rather than a ‘greedy’ algorithm. A ‘lazy’ algorithm is one in which objects are compiled on a need-to-use basis. A greedy algorithm is one in which objects are recompiled as soon as they become out-of-date. A greedy algorithm would perform a lot of unnecessary compilation, because often an object becomes out-of-date yet does not need to be used.

8.5.8 Deletion of Objects

What happens if an object is deleted? Does it cause dangling references in the dependency lists? Here is a typical scenario:

```
int X;                  /* X is created */

int f()                 /* Function f() is created. */
{                       /* It depends on ‘X’. */
     return X;
}

delete X;               /* X is deleted. */

f();                    /* Make() runs on ‘f’. It checks the dependencies.*/
                        /* If ‘X’ has been freed, this will cause all kinds */
                        /* of errors. We really just want an error */
                        /* gracefully reported. */
```

The problem was solved in the following way. Dependency-lists and other information relating to the build tool are stored separately from the object itself. They are stored in a system object called a ‘make-node’ which is linked to the actual object. A dependency list actually consists of a list of pointers to make-nodes. When we delete an object, we free the object itself but its make-node is recycled. Whenever the system examines a dependency from one object to another, the time-stamps provide the system

with enough information to distinguish between a true dependency and a dependency to an object which has been recycled.

Therefore the above code will correctly generate an error, as follows:

```
f();
 <Error: Undefined identifier: 'X'>
```

Furthermore, an object which has not been successfully compiled is marked as out-of-date, regardless of its dependencies. Therefore, we can create an object 'X' and call 'f()' and it will successfully recompile 'f()' with a reference to the new 'X':

```
int X=10;              /* An object called 'X' will be recreated with a */
                       /* new make-node.  */

f();                   /* 'f()' is now recompiled successfully. */
 = 10;
```

If an object is *renamed*, it is deemed to have modified its interface but otherwise it retains the same make-node. Updating its interface-change time-stamp will cause all the objects that depend on it to be considered out-of-date, so they will be recompiled before being used (although probably generating an error unless a new object with the old name is created).

8.5.9 Make() and Classes

If a class (or struct or union) is redefined, i.e. a new definition overwrites an old definition, the two must be checked for equality. Two structs are considered equal if and only if they have the same members with the same names, types and offsets (or the same name and type in the case of member functions). If the class has changed, then the 'interface-change stamp' of that class is updated and everything dependant on it will immediately be considered out-of-date.

8.5.10 The Preprocessor

The C/C++ preprocessor is integrated into Barbados. Macros can alter how a function is compiled in many ways - by inserting braces, by pasting tokens, inserting keywords and so on. However, this does not affect Make(), since very few assumptions are made about the relationship between two successive versions of an object. (The only problem arises if a macro changes the name of a compiled object. In this case, the system will just generate an error "Source-code lost for object X").

If a macro is written to replace some other identifier, e.g. a typedef name, then the macro will overwrite that identifier. This will cause the old typedef's interface-change stamp to be updated. Therefore, all objects which depend on that identifier will be marked out-of-date and will be compiled before being used again.

Conditional compilation, if limited to the inside of a function, is supported with no extra effort by the build tool. '#include' files are not supported in Barbados because they are unnecessary.

8.6 Discussion

8.6.1 Discussion - What is the time-complexity of this algorithm?

There are two elements to this algorithm: traversing the graph of dependencies, and actually doing the recompilations. According to empirical evidence described later, and according to discussion in section 8.6.2, the graph traversal occupies an insignificant amount of processing time compared with the time spent on recompilations. So the time complexity of Make() as a whole can be determined by considering how many recompilations will be invoked.

Let us consider the set of objects (functions, types etc.) whose source-code has been modified. Some of these will have 'interface changes', whereas others will not. In other words, objects will have been compiled directly after being edited, and in the case of functions whose interface doesn't change, there will be no further compilation required. However, if the interface of a function or the contents of a class or struct definition changes, or a function increases in size so much that it needs to be reallocated, then we will require compilations of every object that calls/uses the object. There is no upper bound on this set. Therefore, the time complexity of this algorithm can best be described by:

Make() is in O(a + b), where
a = number of objects without interface changes
b = the sum of all objects which *use* objects which have had interface changes.

8.6.2 Discussion - Does Make() scale to very large programs?

The algorithm used by Make() involves a traversal of the transitive closure (through dependencies) of the program being executed, where this traversal is performed each time code is executed following some modification to the program. It might seem that this graph traversal would be infeasibly slow for large projects which consist of thousands of functions, especially since the aim of an incremental compilation system is to allow the user to make frequent small changes to a program.

For a worst case, consider a menu program which allows the user to select from a menu of large applications - each time the user interactively modifies something, the following call to the menu program will involve a traversal of the dependency graphs for every application.

However, this is not considered a problem for two reasons. Firstly, a program can actually be very large before this becomes a serious problem - a simple graph traversal with infrequent recompilations can be quite quick even for quite large projects. Secondly, Barbados uses large-grained objects (LGOs) to factor out dependencies and dependency checks.

Indeed, the motivation for incremental compilation comes partly from the problem of developing very large applications.

8.6.3 Discussion - Does Make() reach a unique stable configuration?

Let me define a *stable configuration* as a set of program objects which do not change through recompilations of the same source-code.

There is nothing in the Make() algorithm to guarantee that a stable configuration is unique. However, in Barbados with C++, it almost always is. For a counter-example, consider the following case:

```
typedef long A;
typedef A B;
typedef B C;
typedef C A;
```

In this cycle of declarations, the last line overwrites the source for the first line. This cycle will stabilise with all A,B and C all being 'longs', and yet if 'A' had initially been set up as a different type, we would have the same set of source code but with a different configuration.

I do not know of any static set of source-code which through luck or ordering can yield alternative stable configurations.

8.6.4 Discussion - Should Make() be an application or an intrinsic feature?

With a fine-grained build tool, the dependencies are too numerous to have the user specify them, and the invocations of the build tool are too frequent to have the user make them. Therefore it was considered necessary for Make() to be a transparent feature, implemented at the programming environment level.

8.6.5 Discussion - Could the UNIX 'make' be used in the way outlined here?

'make' is the ubiquitous UNIX build tool. Related to it is the 'makedepend' program, which automatically constructs makefiles (or at least makefile dependency lists) given C source files and some knowledge of the C language. One can consider whether the combination of 'make' and 'makedepend' and a C compiler optimised for compiling small sections of code, could give the benefits outlined in this chapter.

However, in practice 'make' cannot be used as a fine-grained build tool. It inherently deals with transformations from one 'file' to another (for example, time-stamps are based on files' UNIX time-stamps).; whereas the objects which a fine-grained build tool deals with are too small for it to be efficient to treat them as files. Furthermore, 'make' has no concept of the distinction between an object's interface versus its implementation, and therefore does not cope with circular dependencies. Also, to properly maintain fine-grained dependency-lists really requires a very close degree of integration between the compiler and the build tool. (Dependencies are very numerous and therefore need to be maintained transparently, they are very dependent on the language definition, and they change very frequently).

8.6.6 Discussion - What Other Languages/Systems would this Apply to?

While this system has only been implemented for C++, there is nothing to prevent it being quite a general solution to the fine-grained build problem. It could be applied to any statically typed compiled languages, object-oriented or otherwise.

To apply this build tool to another language would involve modifying that language's compiler to (a) be efficient at compiling single objects at a time, (b) generate dependency information, and (c) interact with the Barbados persistent store.

8.6.7 Discussion - How does this granularity of compilation compare with the alternatives?

The Barbados compiler and build tool operate at the granularity of individual objects (e.g. classes, functions, variables and macros). That is, dependencies are tracked between these objects and these objects are the units of recompilation. This level is called 'procedure-level incremental compilation'.

There are incremental compilation systems which work at an even finer grain than procedures, for example statements or sub-expressions [Ear72] [Mcc96].

These finer levels of granularity can speed up the recompilation process. However, they have disadvantages: the quality of the generated code can suffer, due to various code optimisations being inapplicable (e.g. code motion, register allocation), and they are probably harder to program. (Although note that procedure-level compilation also comes with a cost, namely that global optimisations cannot be applied. An example of a global optimisation is passing function arguments in registers).

Judging from experience with Barbados, it is not necessary to implement finer levels of granularity. The user seldom notices the pause while recompilations are being performed, even within large projects.

8.6.8 Discussion - How does this solution fit in with configuration management?

Configuration management is the broad problem which includes version control, building applications and supporting different platforms.

The system described in this chapter is not as general as a program such as 'make'. In order to provide the flexibility a user-defined 'makefile' has, a commercial version of Barbados would be able to construct complex software systems by making calls to the normally transparent Make() function. An individual program ('executable') could be developed and debugged with all the benefits described in this chapter, and the application ('set of executables') could be constructed for each configuration more under the user's control.

To support such usage, Barbados would create objects called 'compiler context objects'. These objects would store information such as what compiler was used, what options were used, and what processor was targeted. Each time these options are modified, the relevant code would become out-of-date.

At some stage, users would also like to be able to take copies of programs or applications, e.g. to release the software or have stable backup versions of the application. A utility is planned for Barbados which will use dependency information to wrap up functions and their transitive closures into a single object which can then be regarded as an executable or object file. This action will effectively unlink the object code from the source-code.

8.7 The Consequences of Fine-grained Builds

When a system provides an interactive programming interface such as Barbados, and a fine-grained build tool, the consequence is that incremental compilation and interactive compilation are provided.

> **Definition** "*Incremental Compilation*": Incremental compilation refers to programming environments where a small modification in source-code will generally lead to a correspondingly small amount of recompilation in order to bring a program up-to-date.
>
> **Definition** "*Interactive Compilation*": Interactive compilation refers to programming environments with a very short edit-compile-debug cycle.

Incremental compilation is not a well-used technology. One possible explanation for this is that until now, the fine-grained build problem has not been recognised or solved. Existing approaches to incremental compilation are described in the following two sections.

Interactive compilation provides the user with the best of both worlds of compilers and interpreters, namely:

- Fast execution of code (a property of compilers)
- A quick edit-compile-debug cycle (a property of interpreters)
- Early error detection (a property of compilers)

A consequence of providing interactive compilation within an IPPE is 'monolingualism'.

> **Definition** "*Monolingual System":* A Monolingual system is one in which the high-level programming language, and the shell-level programming language are combined into the one language.

In almost all existing programming systems, there is one language for high-level programming tasks and another language for invoking applications and interacting directly with the directory hierarchy and operating system. However, in an IPPE there is no distinction between the directory hierarchy and a process address-space, nor is there a distinction between functions and applications. Therefore, with the provision of interactive compilation the system becomes simultaneously a high-level programming environment as well as a shell-level operating environment.

Advantages of a monolingual system include:

- Users need only become expert in one language
- In a monolingual system, the barrier between shell-level objects and high-level programming language objects is broken down, making communication between the levels easier.

There have been attempts to provide monolingualism before, [Hee85][Fra83]. However, monolingualism has not yet become popular. This is

perhaps because a monolingual system needs to solve the persistence problem first, and this has not been done before outside the context of IPPE's.

8.8 Results

My system was compared against two other development environments. The experiments were based on comparing how many lines of code each compiler had to compile in order to bring a program up-to-date. A single large application was used:

Test Application	'tt' : A program for generating high-school timetables using AI search techniques
Size and statistics	38 modules, all written in C; 38 header files. 20159 lines of code in *.c files 1161 lines of code in *.h files

This application was not originally written for Barbados, and it was not modified in any substantial way for Barbados. It was chosen as the test application on account of these facts and its size. There is a high degree of interaction between the modules, i.e. it has a relatively dense dependency graph. There is a central header file of about 500 lines which contains a number of mutually recursive type definitions and which is included by almost every module.

The environments compared are described below:

	Barbados	*gcc v2.7*	*Visual C++ v4.0*
Description	The system described here	The gnu C compiler, common on UNIX systems, together with the UNIX 'make' command	The Microsoft C/C++ integrated development environment; supposedly a state-of-the-art incremental compilation system
Method of counting # of lines compiled	A special function was added to count lines of source code as they're compiled	The 'makefile' was extended to pass input files through the preprocessor and then through a line-counter.	Analysis of the 'build window' messages and a bit of guesswork based on the descriptions of compilation techniques given in the documentation.

In Barbados, the preprocessor is integrated into the lexical analyser. The 'gcc' preprocessor has the property that it outputs the same number of lines as are input through include files and source files, which means that the number of lines that pass through it can be compared with the number of lines passed to the Barbados compiler.

In Visual C++, the system informs the user of which source files are being analysed or compiled. This information, along with the compiler documentation's description of the incremental compilation techniques, was used to calculate how many lines of code were probably compiled.

Various scenarios, consisting of small changes to an application's source-code were tried out in each environment. The number of lines of code each compiler needed to compile are listed below.

(The lines-of-code metric was chosen for the comparison in order to compare the effectiveness of the build tool rather than the speed of the relevant compilers. Multiple numbers are given where multiple examples were tried).

Lines of code	*Barbados*	*gcc*	*Visual C++*
To rebuild all modules	20159	150231	21320
Adding a comment to the central header file	4	139736	34926
Adding a comment to a type-definition in the central header file	30,37	139736	1107 lines of analysis (?)
Adding a member to a struct in the central header file	6371, 5448, 6173	139736	815, 2329, 815 if added to the end of struct; 26533, 27822, 26533 if added to start of struct
Adding a function prototype	4	139736 with the central header file; 21333, 11402, 25919 with peripheral header files	34926 with the central header file; 9681, 3229, 7594 with peripheral header files
Adding a parameter to a function	539, 380, 326	12699, 13823, 53272	3662, 5076, 18139
Changing the implementation of a function	9, 89, 48, 31 for small changes; 310, 336, 51, 34 for large changes.	3446, 2952, 3651, 3329	9, 89, 48, 31 with padded object files

These figures show the (often) vast differences between Barbados and the other environments.

The traditional environment, represented by 'gcc+make' suffers because (a) each module must separately compile large amounts of header files and (b) because large units of code must be recompiled when each change is made. The standard C header files comprised about three quarters of the above figures. Of the remainder,

about half consists of source files (*.c) and half consists of header files specific to that project.

The Visual C++ environment has the following features available to speed recompilation: (a) precompiled headers, (b) incremental compilation of function bodies, (c) 'minimal rebuild' (described below) and (d) incremental linking. All features were turned on for the experiments, except for precompiled headers of application header files. These features are detailed in [Mic96]. Interestingly, features (a) and (b) are turned off by default. The reason given was that the cost of maintaining the necessary information is generally greater than the cost savings for small and medium-sized applications. The same is not true for Barbados: in Barbados, the cost of compiling and building a program increases approximately linearly with the size of the program.

The 'minimal rebuild' feature is the closest feature to the techniques described in this chapter. It consists of maintaining a database which relates each source file to the class interfaces given in each header file. If something changes inside a class interface, it is these 'class' dependencies which are checked with each source file rather than the normal dependencies linking source files with header files. (The application had to be compiled as C++ source for this feature to be turned on). However, changes made to header files outside class/struct definitions are not dealt with by this optimisation and they often cause large amounts of recompilation. Such changes can include adding comments, declaring new functions or objects, or adding new classes. (Admittedly, good C++ header files should consist entirely of class definitions, but nevertheless this technique would be a lot more useful if it were more robust).

Visual C++ has an incremental linker and a full linker. The environment supposedly uses the incremental linker in most situations but occasionally needs to resort to a full link. Barbados does not require a separate linking phase; this is performed as a by-product of compilation.

Even very large functions can be recompiled in Barbados without a noticeable delay, e.g. 300ms for a 500-line function. This result vindicates the decision not to explore finer grains of incremental compilation.

8.9 Literature

The closest work to this system is probably the LOIPE system [Med81]. LOIPE was an integrated programming environment for a language called GC, designed for LOIPE, which was a type-safe version of C. This system was quite advanced for its time. It consisted of a number of tools, all of which interacted via a syntax-tree representation of programs (including the editor). LOIPE was part of the Gandalf project [Not85]. While this system is probably quite efficient at reducing compilation (although empirical comparisons do not seem to be available), the paper did not describe how the system deals with the various subtleties described in section 9.3. Perhaps the language implemented was a very limited subset of the C language (the language was defined in a technical report).

Incremental compilation occurs naturally within an integrated development environment. A number of incremental compilation environments exist, for example Orm [Mag90] [Gus89], Gandalf [Not85], The Synthesizer Generator [Rep85] [Tei81], PSG [Bah86], Mentor [Don80].

Most of these systems are based on formal language definitions of one kind or another. For example, the Synthesizer Generator uses attribute grammars, Mentor uses 'METAL': a 'meta-language' for specifying languages, Orm uses 'Door Attribute Grammars' (an extension to attribute grammars) and PSG involves a custom-designed 'formal language definition language'. My system is different from these because it is based on very standard compilation technology. That is, it achieves incremental compilation using new 'build' technology rather than using new compilation technology. As a consequence, Barbados does not suffer any of the limitations of having to express a language formally: formal language definitions can suffer limitations such as: (a) not being able to express all classes of grammars, (b) having trouble with complex naming schemes and cross-module links, (c) not being able to handcraft the compiler for better performance or intelligent error-reporting.

All of the above systems perform incremental compilation by manipulating syntax-tree representations of programs. In fact, text versions of program source are not even stored, since the editors, compilers and debuggers in these systems all use the syntax-trees. This property makes it possible to implement very fine grain incremental compilation, however it also reduces the scope for optimising the generated code. None of the papers reviewed discuss (explicitly) issues such as propagating changes in type definition throughout programs. The Synthesizer Generator and its precursor, the Cornell Program Synthesizer either make it a non-issue by virtue of the fact that they interpret the syntax trees directly, or they propagate such changes greedily. PSG has a 'lazy compilation' algorithm, whereby the modifications to a syntax tree's attributes are made when execution reaches that point. By comparison, Barbados manipulates ordinary text representations of programs, rather than syntax-tree representations. This property is an advantage for users who prefer to use standard text editors to create programs, (although the PSG editor does have a 'textual mode'). Linear text is also a much more compact representation. Barbados performs procedure-level incremental compilation, which means that traditional code optimisation techniques can be used. Also, compilation (and therefore error-reporting) occurs just prior to execution of a program. I would argue that this is preferable to propagating all changes as soon as modifications are made to a source object, because such a 'greedy' algorithm would perform much unnecessary processing of code. It is also preferable to compile objects/statements before a program runs, rather than as execution reaches them, because the user generally wants to deal with errors before run-time.

Of the above systems, Orm, the Synthesizer Generator, PSG and Mentor are based on statement-level or expression-level incremental compilation. The disadvantage of these systems is that the opportunities for the compiler to allocate registers and optimise code is considerably reduced. The advantage is that more incremental compilation can be achieved.

As a result of the fine-grained build approach, it has been possible to implement a reasonably rich subset of the C++ language in Barbados. The current implementation supports the preprocessor, inheritance, overloading (and some persistence extensions which would tax any language definition language). Furthermore, a side-by-side comparison of my system with other program development environments has been performed, with a large and real software project, to demonstrate the relative amount of code sent through the compiler; with good results.

There are several books and papers on the UNIX 'make' program and its variants, including the original paper by Feldman [Fel79], plus [Ora91] and [Dub93].

However, as discussed above, the UNIX 'make' does not address the problems of fine-grained builds.

A relevant paper is "Smart Recompilation", [Tic86], which discusses a method for minimising compilations of modules by processing source files to separate interfaces from implementations. This work is also in the context of traditional systems and compiling at the 'modules' level. Another paper of some relevance is "Lazy and Incremental Compilation" ([Hee94]) which discusses programs that compile fragments as execution reaches an uncompiled fragment. This concept of 'lazy compilation' is an attempt to reduce compilation times by a very different method to ours. While it is a worthwhile avenue to investigate, it could be a very difficult problem in general.

Various programming systems and integrated development environments have their own built-in build tool, e.g. all Ada and Eiffel systems. The 'Eiffel' tool shares some similarities with the build tool as described here[5], although the basic algorithm does not generalise to a language such as C++. Unfortunately, there do not seem to be any published sources describing the details of Eiffel's build except [Mey88].

8.9.1 Related Problems

Related problems include the problems of version control, configuration management and data evolution.

A version control system is a mechanism for storing and retrieving different versions of source-code (or other program components). One example is RCS [Tic85].

Configuration Management is discussed in 9.6.7.

Data evolution is the problem of dealing with existing data as type-definitions (and also the semantic interpretation of fields) change. For example, if a field is added to a 'struct', there might be a lot of existing instances of the old version of that type - and the user might need them to be updated. The solution to such a problem could benefit from ideas presented in this chapter, however I have not attempted to solve this problem. It is a remarkably difficult problem to solve generally, since it can involve the human problem of managing data at remote sites, and leads into problems of semantic changes to data-structure representations and invariants. Practitioners work around the problem using techniques such as 'self-defining messages' or objects with fields tagged by their definitions.

8.10 Summary

Barbados implements 'fine-grained builds'. It is necessary to provide this feature before an IPPE can be an effective programming environment.

As a consequence, Barbados solves the problem of incremental and interactive compilation. Fine-grained builds constitute a novel solution to this problem.

---

[5] With Eiffel, there are two kinds of dependencies - interface dependencies and implementation dependencies. A client of a class depends on that class's interface, whereas a child of a class depends on that class's implementation. In Barbados, entities depend separately on member functions; and private and public members are lumped together for the purposes of dependencies; so such a distinction is not necessary.

The fine-grained builds solution is simpler, more general and more efficient than competing incremental compilation technologies.

9. Description: Barbados Programming Tools

Intrinsic to the Barbados programming environment are the editor, compiler and make tool. In addition, there are a number of 'tools' which are invoked as ordinary Barbados functions (objects).

For example, there are commands to rename and delete named objects plus their source-code, and to retrieve the source-code of specified objects.

There is a 'grepdepend' command, which searches for objects which depend on a specified object.

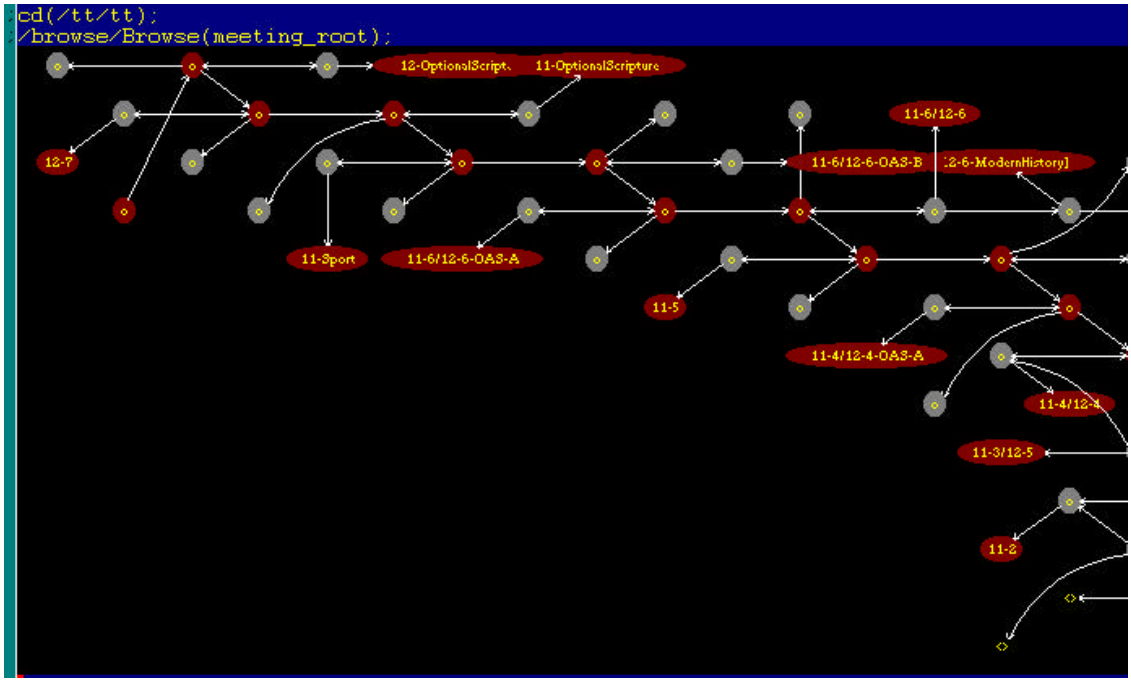
The Barbados '`Help`' command takes a specified class or instance, and lists all the available operations on it, for example:

```
Help(Bitmap)
class Bitmap {
     Bitmap(int w, int h, int ncol);
     Clear();
     Clear(int col);
     SetPixel(int x, int y, int col);
     Line(Place A, Place B, int col);
     ...etc...
};
```

The '`Help2`' command also examines source-code for the specified class and brings up comments associated with each member function. The command extracts the 'synopsis' for the specified object, where the 'synopsis' is defined as a contiguous block of comments between the function prototype and the definition. For example:

```
Help2(Matrix)
class Bitmap {
     Matrix(int n, int m);
     /* Creates a blank matrix */
     Matrix Transpose();
     /* Returns the row-column flip Matrix */
     Matrix Identity();
     /* Sets it to I. Must be square. */
     operator*(Matrix M);
     ...etc...
};
```

The 'Browse()' command displays a 2-D rendering of a data-structure, (something rarely seen except on whiteboards). For example:



This display shows a real-life data-structure as used by a program called ‘tt’. ‘tt’ is discussed in Chapter 11. This data-structure is a linked list with associated objects, and can be seen extending from left to right across the above diagram.

This raises the issue of whether the type information should be used to display data-structures with their high-level structure more obvious. For example, perhaps the above linked list should be displayed in a line with the associated objects extending above and below it.

Incidentally, this picture also demonstrates the persistence feature of Barbados. This data-structure resides on disk in this format, and the above commands show the user entering the appropriate directory for the first time during this session and finding the data-structure there without needing to execute any code.

10. Persistence in Barbados

10.1 Introduction

The Barbados persistent store is a two-level persistent store: this means that there are two granularities of objects.

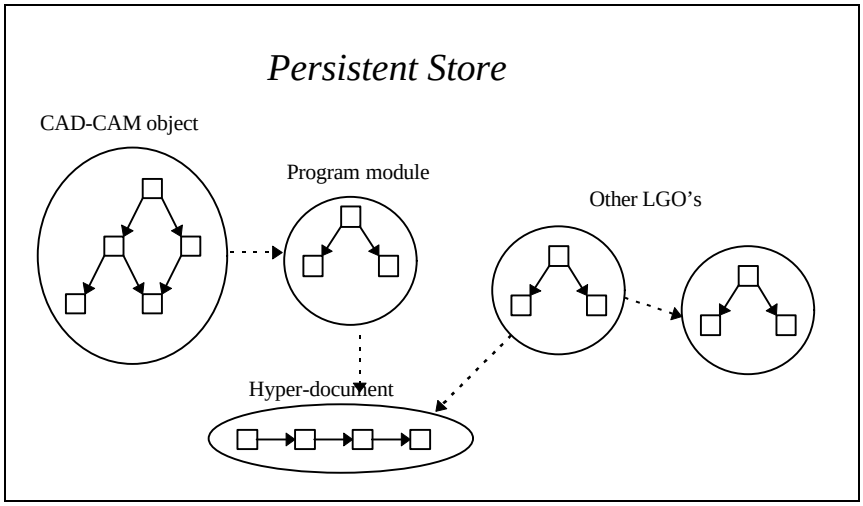
Definition *"Fine-grained object":* A 'fine-grained object' is a small, typed object such as an integer, class instance or function.

Definition *"Large-Grained Object (LGO)":* A 'Large-Grained Object' is a collection of fine-grained objects. The fine-grained objects must each be reachable from a special fine-grained object called the 'root object' of the LGO. Large-grained objects can be thought of as having a type defined by the type of the root object.

Any persistent system will need to deal with fine-grained objects. Barbados was designed to deal with large-grained objects as well, because LGO's provide access control (permissions checking, memory protection, locking and sharing) which cannot be provided effectively at the fine-grained level, and because LGO's give programmers and applications a convenient intermediate structure to work with.

An LGO can be thought of as a data-structure. Typical LGO's could include a program module, a word-processor document, or a CAD/CAM subassembly.

Higher-level data-structures can be constructed by having LGO's point to each other. For example:



LGO's are migrated in and out of memory in a single transaction and are stored in a single location. Pointers inside these objects are swizzled as the large-grained object moves in and out of memory.

A consequence of this system is that these large-grained objects become 'portable data-structures', i.e. they can be ported between address spaces. This

property may be useful in implementing concurrency across distributed architectures (e.g. with transputers).

The advantages of this form of persistence are (a) having access control over convenient-sized groupings of data, (b) having a standardised, enforced intermediate grouping of data suitable for applications and system tools, (c) being able to easily copy / move / compare structured data between large-grained objects, (d) having processing performed about as efficiently as in traditional systems.

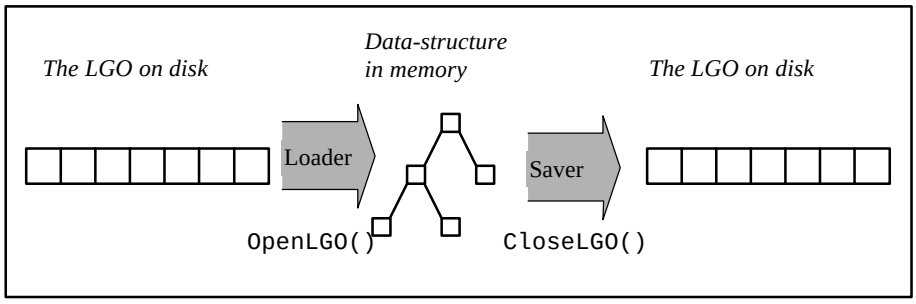
The Barbados persistent store was designed to support intensive computation on data-structures, i.e. optimised for the schema: read-in-data-structure, process-it, write-it-out. It was not designed to act as an effective object-oriented database, and more work would be required before Barbados could claim to be an OODB.

10.2 Persistence as seen by the User

10.2.1 Load-Process-Save

Computation occurs by reading one or more Large-Grained Objects (LGO's) into memory, operating on them, and then writing them out again. This mapping stores the LGO on disk in a compact form and in memory in a structured format, i.e. as a data-structure with ordinary main-memory pointers.

These two mappings have to be explicitly invoked, through an 'OpenLGO()' and 'CloseLGO()' call (or through functions which in turn call 'OpenLGO()' and 'CloseLGO()'[6]). However, once invoked, they handle the mapping from disk format to memory format automatically using type information. This is illustrated by the following diagram.



In this sense, the persistence is not orthogonal. It is argued that these calls will be made relatively infrequently, and that they will occur at logical points in the program e.g. when the user opens a new document, and also that semantic benefits accrue from making this call explicit (see the section on the argument for LGOs).

[6] The 'chdir()' function to change the current directory is a good example. It closes the LGO of the old directory and opens the LGO of the new directory.

10.2.2 Creation of LGOs

In the Barbados type system, (which applies across all languages), there are two types of pointers: ordinary main memory pointers and LGO pointers. The ordinary pointers apply to the current process's address space, and the LGO pointers are persistent identifiers which are used to identify LGO's on disk.

LGO's are created with the CreateLGO() function. The CreateLGO function simultaneously creates a fine-grained object and an LGO. Pointers to both entities are returned: the pointer to the fine-grained object is returned as the formal return value, and the pointer to the LGO is passed back when an LGO pointer is passed by reference to the function call. The fine-grained object is the root of the LGO. (The CreateLGO() function replaces the 'new' operator when the programmer wants to create a new LGO root object.)

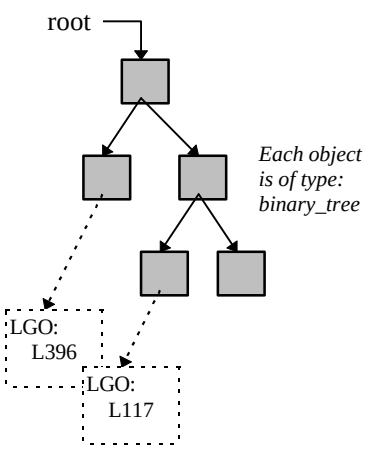
The LGO is created as an 'open' LGO, meaning that (a) it must later be closed, with the 'CloseLGO()' function, and (b) it is associated with its own 'heap'. The programmer must use this heap to store the fine-grained objects which they wish to belong in this LGO.

At any point there can be multiple open LGO's. Therefore there are multiple heaps simultaneously existing. The user must specify which heap an object is to be placed in whenever they dynamically create a fine-grained object (i.e. using 'new' or 'malloc' or other functions). The user does this by either: (a) putting the LGO pointer as an extra parameter to 'malloc' and 'new' function calls, or (b) calling the usual functions without an extra parameter, which will cause the system instead to use the 'default heap'.

The default heap can be set at any time by calling the 'SetDefaultHeap()' function. It takes one parameter, namely the LGO pointer of the desired heap. However, it is also set on each call to OpenLGO() or CreateLGO(), to the LGO being accessed or created. This means that often the ordinary forms of 'malloc' and 'new' can be used, which is beneficial particularly because it means that existing source-code can be ported easily. This technique is especially useful when used in conjunction with the 'cd()' command. The 'cd()' command, when used to change to a new directory in a different LGO, contains an implicit call to OpenLGO() and therefore 'SetDefaultHeap()'.

The CreateLGO() is actually an operator defined as a language primitive, because of the way it manipulates types. CreateLGO() requires an LGO pointer as a parameter. The type of this pointer is extracted and used to determine what type the LGO is to be. This information is needed to determine the size of the fine-grained object which is created. The formal return value of the CreateLGO() function is the address of the fine-grained object, and its type is constructed by converting the type of the parameter from an LGO pointer to an ordinary pointer. If Barbados implemented the C++ 'template' mechanism, these functions could instead be provided using templates.

To manipulate a large data-structure, which cannot fit effectively into one LGO, the user 'breaks up' their data-structure by inserting LGO pointers rather than ordinary pointers into their type-definitions. I believe that information can generally be organised into relatively autonomous 'sub-systems' which can act as LGOs, i.e. that there are generally very natural places to break up a data-structure into LGOs. Often one LGO is sufficient for a data-structure.



Creation of an LGO: This data-structure, at least the grey portion, can be saved as an LGO using the following code:

```
binary_tree lgo TreeSeg;

. . .     // Creation of tree

TreeSeg = CreateLGO(root);
```

10.2.3 Address Spaces

A process in Barbados always owns (exclusively) an address space. The reason for this systemis to provide memory protection.

An LGO on disk is effectively contained within its own address space. (All pointers are represented as offsets from the beginning of the LGO). When LGOs are opened, they are read into memory and they have their pointers swizzled so that they belong in the address space of the relevant process. Hence an LGO can be thought of as a *portable data-structure*, in the sense that it can be ported automatically between different address spaces.

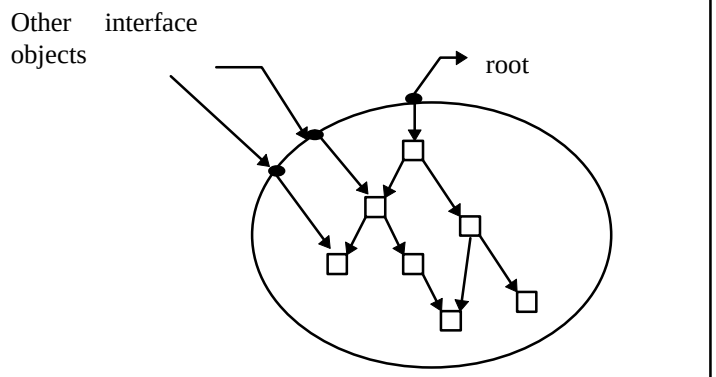
In order to merge or compare data from multiple LGOs, the user opens the various LGOs simultaneously and then accesses them as local data-structures.

Most other persistent systems do not have this ability to move data around between address spaces. This ability makes it easy in Barbados to transport data between different processors or to remote systems.

10.2.4 Information Hiding vs Referential Transparency

An LGO acts as an information-hiding object - each LGO has a persistent identifier, but the small objects inside do not (except for ‘interface objects’ - see ‘LGO-Name Swizzling’).

An LGO, showing internal and external references



The reasons for this are twofold. On one hand, it makes the implementation easy and efficient. On the other hand, it was considered desirable to provide this form of information hiding/protection.

For example, use of this kind of information hiding means that the system has complete control over the organisation of data within a LGO. Disk space is not wasted on freed memory - LGOs are completely compact on disk. When the user requests a certain piece of information off disk or across the network or even from another processor, that information is provided in the most compact possible form. There is also the potential for transparent system-level data-compression of LGOs, essentially because of the fact that random access is not needed within an LGO while on disk.

10.2.5 Memory Protection

The Barbados Persistent Store has been designed to support type-unsafe languages, such as C++. Therefore, an alternative form of memory protection to type-safeness is required.

With the current system, the user can only modify the store in the following way:

1. Provide a valid LGO identifier. LGO id's come from a sparse name-space, i.e. a random value is unlikely to be a valid identifier.
2. Open that LGO in write mode. The user must have write permissions on that LGO, and furthermore that LGOs type must match the type the user is expecting.
3. Follow the pointers within that LGO to reach the desired small object(s).
4. Close the LGO.

This protocol ensures that bugged programs do not corrupt the entire persistent store. A bug will generally only propagate to those LGOs which the process currently has open in write mode, and it can never propagate to regions where the user does not have the required permissions.

In this sense, memory protection in Barbados is similar to that of traditional (e.g. UNIX) systems.

It should also be noted that orthogonally persistent systems cannot provide this particular form of memory protection. Without explicit large-grained objects, there is no concept of the programmer informing the system of which regions of the persistent

store are going to be modified. Therefore, even type-safe operations could make unintended modifications to the persistent store. Also, in order to delete information, the user must explicitly delete an LGO - not just 'drop' or overwrite a reference.

10.2.6 LGO-Name Swizzling

A directory hierarchy is provided in Barbados by adding a new fundamental type, namely 'directory', to the Barbados type-system. A directory is a collection of 'named objects', where a named object is a tuple consisting of: (name, type, storage-class, value).

According to the information provided so far, function objects in different LGOs would not be able to reference i.e. call each other - except by opening up the LGO, finding the directory containing the called function and looking up the name in the directory.

Because this would be a very inefficient way of linking code together, an optimisation called LGO-Name swizzling (L-N swizzling) has been provided.

L-N swizzling deals with directory LGOs, i.e. LGOs with directories at the root. In effect, it allows objects inside LGOs to reference objects inside other LGOs by providing the name and LGO id.

It works as follows:

How L-N handles are represented:

- Any object inside the directory at the root of an LGO is automatically designated an 'interface object', (unless it is declared as 'private').
- That object can be identified persistently by a (LGO id, name[7]) pair.
- When a LGO is stored on disk, there is room for an 'L-N table' at the end of the data. This table consists of a sequence of (LGO id, name) pairs. Any pointer inside the LGO which points into this table is taken to mean a reference to that named object in that LGO. (This can be called a 'foreign object').

How the Loader deals with L-N handles:

- When that LGO is read into memory, each pointer is swizzled. If a pointer is an L-N table pointer, the system links in the specified LGO and then further swizzles that pointer until it points directly to the denoted object.

How the Saver deals with L-N handles:

- When an LGO is being saved onto disk, pointers to foreign objects must be detected.
- Each LGO belongs in a separate heap in memory, and therefore it is possible to map an arbitrary pointer to an open LGO. In this way, it is possible to know whether a given pointer points into the same LGO or into the main process heap (in which case it is dealt with normally), or if it points into another LGO.
- If a pointer points into another LGO, the system determines whether that object is an interface object or not. If not, then that pointer value is set to NULL and an error

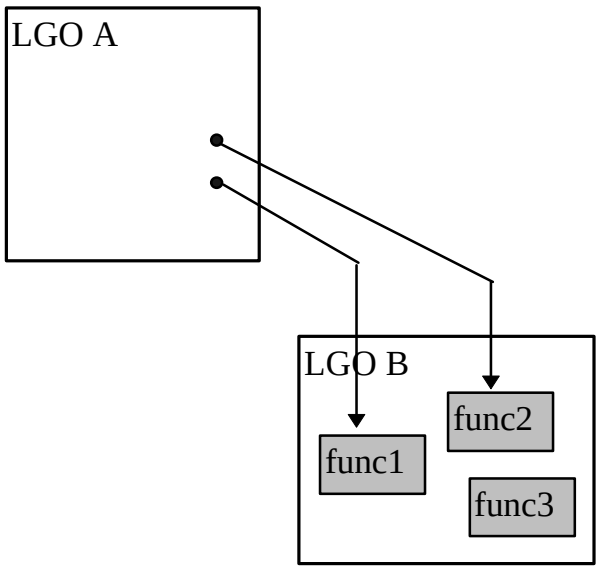
[7] L-N swizzling extends to member functions of an interface class. They are given names such as: 'class1::fn1'. Also, overloaded objects have unique names constructed for them by appending version numbers to the name, e.g. `Print'4`.

should be reported. Otherwise, the (LGO id, name) pair is constructed and added to the end of the LGO.

As a consequence, all function-to-function references (i.e. function calls) are swizzled at load-time, so that during run-time they impose no overhead whatsoever.

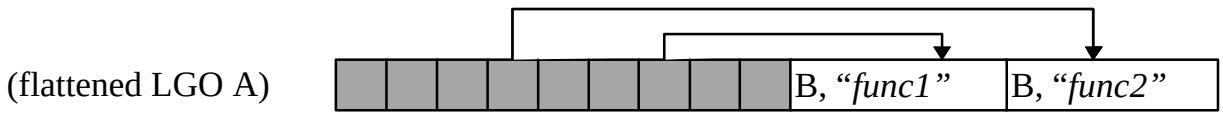
In addition, type objects and data objects can be referenced via L-N swizzling. This situation is in fact quite common. Suppose for instance that the user writes a complex number class in the "/cl/complex" directory, which resides in its own LGO. The type-definition for a complex number would then come from this LGO. Since the 'complex' class object is at the root of this LGO, any pointer to it from an external LGO will be L-N swizzled. The actual type information will therefore not be stored in any of the client LGOs but only in the 'complex' LGO. This property would not be much of a saving until one considers that a 'type definition' includes all the code for the member functions, and that we would not want this information copied into each LGO.

One small detail which has been touched upon is the concept of 'linking LGOs'. If one LGO references another via L-N swizzling, then it will require that other LGO to be loaded into memory whenever it is in memory itself. The 'open-LGO manager' subsystem of Barbados therefore allows LGOs to be loaded either from an 'OpenLGO()' call or by a 'LinkLGO()' call. If one LGO 'links' in another, i.e. it 'depends' on another LGO, then the open-LGO manager will always ensure that the latter is loaded if the former is loaded. It will also close the 'needed' LGO as soon as it is no longer needed (e.g. the requesting LGO has been closed).



L-N swizzling:

If the root of B is a directory, and there are specified 'interface' objects func1, func2, func3 in it, then these can be referenced directly using 'L-N swizzling'.



S-N swizzling provides a limited form of inter-LGO references. I believe that arbitrary inter-LGO references (i.e. one fine-grained object referencing another in another LGO) should not be allowed, on the grounds of information hiding.

Under the present scheme, only *named objects in the directory at the root of an LGO* are eligible for L-N swizzling. This raises the issue of whether Barbados should generalise the concept of what is eligible to be an interface object.

On one hand, it might seem more flexible for arbitrary objects to be capable of 'interface' status. On the other hand, it could be argued that if an object is important enough to be an interface object, it should be important enough to deserve a symbolic name and listed somewhere (e.g. the directory at the root of the LGO). Putting an object in the LGO root-directory, (under a meaningful name or even a computer-generated identifier) could be seen as a method of 'registering' an object as an interface object, in which case the current scheme can be seen as a general scheme.

10.2.7 Deep Copy and Duplicating Objects

A consequence of this form of persistence is that it becomes easy to perform deep copies of data-structures, because the LGO boundaries provide meaningful limits to the copying operation.

For example, if the user takes a copy of an LGO which contains pointers to things such as the persistent root (the root directory) or the compiler or font tables, they will receive copies of all fine-grained objects inside that LGO but not the root directory or directory or fonts etc.

10.2.8 Sharing and Locking

There is a multiple reader/single writer protocol with LGOs. In other words, a given LGO is either (a) available to be read by any process, or (b) available to be written by a single process.

10.3 The Barbados type system

One feature of the C/C++ type system which some other persistent languages do not share is that the fields of a structure can be either other structures or pointers to structures. (Member objects can be either values or references). That is, it is possible to directly 'nest' or 'embed' structures inside structures, for example:

```
class vector {
      float x,y,z;
};

class Camera {
      vector Position, Direction;
      matrix M;
};
```

This feature is useful. Where it is used, it is more efficient than the alternative of creating separate heap objects per sub-struct. It can also involve less work on the part of the programmer.

The developers of Eiffel have seen fit to introduce this feature as 'embedded objects' into the Eiffel language. However, many other languages (e.g. Java, Napier) do not support it.

For instance, suppose one creates a C++ type which consists of an array of 100 elements of type 'vector' as defined above. To create an instance of this type, it is sufficient to declare it. In other languages where objects cannot be nested in this way, the programmer must write a loop to create the 100 instances of type 'vector'. Furthermore, the system must make 100 calls to the heap routines to create these objects.

10.3.1 Garbage Collection and Deletion

Large Grained Objects (LGOs) are essentially groups of small objects. LGOs are deleted explicitly, which means that dangling references are possible, but LGO identifiers are not often reused - so accesses to dangling references will almost always be trapped.

Small objects are deleted either in the normal C++ way, (using 'free()' or 'delete') or by being left behind when a LGO is closed and the Saver is called. In the case of the latter, the algorithm is essentially garbage collection by reachability. The Saver only saves those objects reachable from the LGO root, and since the LGO's pages are re-used following the call to the Saver, any other object will disappear from the system.

It was considered that with small objects, the user is unlikely to want to bother with explicitly deleting each object that needs to be deleted. However, with the LGO - a large package of related information, it was considered that the data should not be lost without explicit directions from the user. This form of object persistence/garbage collection is the same as used in the MONADS system [Ros85].

The fact that garbage-collection is provided in this very limited form was partly designed as an optimisation (disk & memory space can be saved by performing garbage collection), and partly arose as an accidental consequence of the 'Saver' algorithm. The 'Saver' algorithm swizzles pointers to their on-disk format by performing a top-down transversal of everything reachable from the root of the LGO, and garbage is never reached by this algorithm.

10.3.2 Persistence vs Deletion of Large-Grained Objects

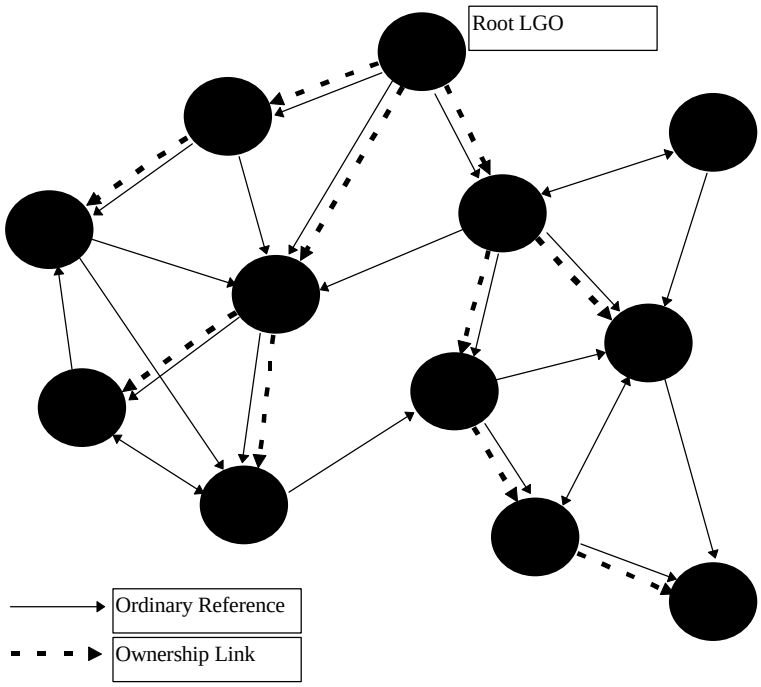
Large-grained objects must be explicitly created and deleted. The 'CreateLGO()' function creates an LGO of a given type, and the 'DeleteLGO()' function deletes an LGO (if possible).

LGO's are organised into a strict hierarchy. Each LGO has one and only one parent LGO, except the root LGO. The parent LGO is specified as a parameter to the CreateLGO() operator. The functions which create, delete and move LGO's maintain this strict hierarchy. For example, it is possible to move an LGO to another part of the hierarchy, i.e. change its parent, however it is not possible to move an LGO to be a child LGO of one of its descendants.

The DeleteLGO() operation will fail if the specified LGO has child LGO's. The child LGO's must be deleted first. However, there is nothing to stop the user from writing a function on top of DeleteLGO() which recursively deletes LGO's.

Any LGO can reference any other LGO, since this merely requires that the first LGO contain a value of type 'LGO pointer to type X'. In fact, these values can be copied and passed around indiscriminately. They can even be put through a data-compression program and reconstructed. However, these references are separate from

the references which form the LGO hierarchy. (Let us call these LGO ownership links). The LGO ownership links are not so easily manipulated.



The above diagram shows the two types of references: 'ordinary references', i.e. an LGO pointer, as an ordinary value in Barbados's type system, and 'ownership links'.

The ownership links are kept in a database maintained by the system. They can only be accessed using special system calls. The tools which Barbados provides to the user, e.g. the "mkdir()" function, attempt to maintain a correspondence between ordinary references and the ownership links, so that the ownership links are a *subset* of the ordinary references. However, if the user chooses to delete ordinary references to child LGO's, this 'subset' relationship will be lost; one LGO will have an ownership link to a child LGO and yet lack an ordinary reference to it.

The user should avoid such a situation, since the environment is geared toward using ordinary references to do everything, and it is cumbersome to use the system functions to access ownership links. (The ownership links are only there as a form of security, to guarantee that the LGO's form a strict hierarchy). If this situation does occur, the user can either (a) use the system calls to restore ordinary references within the user's data-structure, or (b) in the case of directory LGOs, the 'RepairLGO()' function will create named objects in the directory at the root of a LGO, which provide ordinary references to 'lost' child LGO's.

10.4 Persistence Implementation

10.4.1 Pages

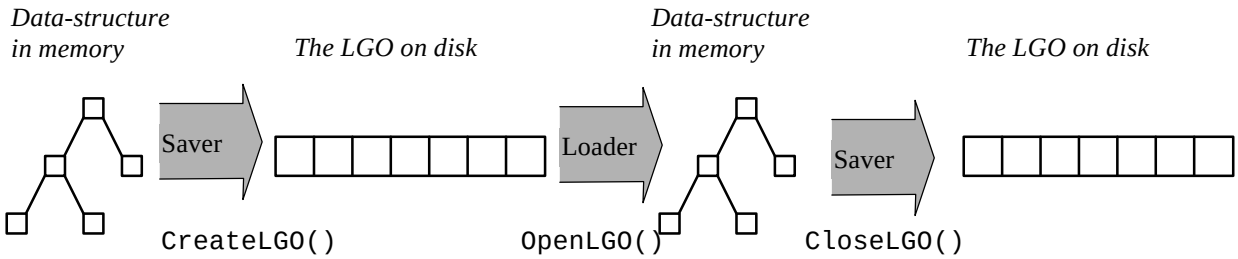
One of the *least* important aspects of the Barbados system is the concept of 'pages'. Pages are very useful at organising memory and disk. Barbados in fact has two independent page systems: there are disk pages used for organising the persistent store,

and there are memory pages used for organising address spaces and heaps. The two page sizes may even be different.

Pages are considered an unimportant aspect of Barbados because they are concepts local to the disk-access subsystem and to the heap-management subsystem which have not had much effect on the design of the rest of the system. The interfaces to these subsystems just deal with variable-size sequences of bytes.

10.4.2 Algorithms used in disk-memory mappings

The function which copies a data-structure to disk is called the 'Saver'. The function to copy it back into memory is called the 'Loader'. 'CloseLGO()' calls the saver, and 'OpenLGO()' calls the loader.



The saver operates in two passes. In the first pass, it identifies all small objects reachable from the LGO root (via memory pointers). It also determines what offsets each object will have in the on-disk LGO. (An object's position in memory is completely independent of its position on disk). In the second pass, the saver writes each object to disk in the order it reached the object, after establishing what offset to swizzle each pointer to.

Usually there is a correspondence between C++ objects and heap blocks (a heap block is a block of memory as created by 'malloc()' or 'new'). If there is any discrepancy, e.g. the block size does not match the object size, the saver deals only with heap blocks. For example, the result of the C++ call: 'strdup("Hello")' (which allocates 6 bytes and copies the given string into it) is a valid memory block and will be saved as it is.

The loader has just one pass. It reads the LGO into memory verbatim and then traverses it, swizzling each pointer from an 'offset' to a pointer relative to the current address space. The information on disk is deliberately formatted into a form compatible with the Barbados heap. That is, each small object is preceded with the appropriate header for heap blocks. This system simplifies the loader - it does not need to move any data around inside memory, it just adjusts pointers.

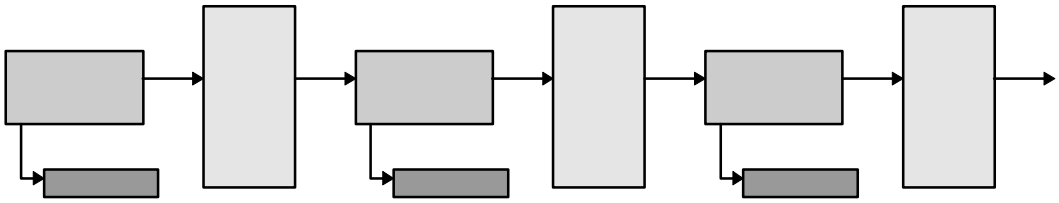
For the Saver to work, the data structures must match their type-definitions. If a pointer field of one type points to an object of a different type, the Saver will usually fail. The user is allowed all the (dubious) liberties that C++ allows, *temporarily*, but at the time of a Saver() operation all type information must be consistent.

Incidentally, this means that if an LGO is 'closed', then it is type-consistent, i.e. can be used directly by a type-safe language without violating type security.

'void*' pointers are dealt with. Whenever a 'void*' pointer is reached, the given block is set aside to be saved, however it is not recursed down until proper type information is available from some other pointer.

An optimisation is provided in the Saver so that linked-lists are flattened to disk by a loop rather than by recursion. This was done so that the stack does not overflow with long linked-lists. The optimisation involves looking for small objects containing just a single pointer, or objects with just one pointer to an object of the same type.

This optimisation can be fooled, e.g. with the following data-structure:



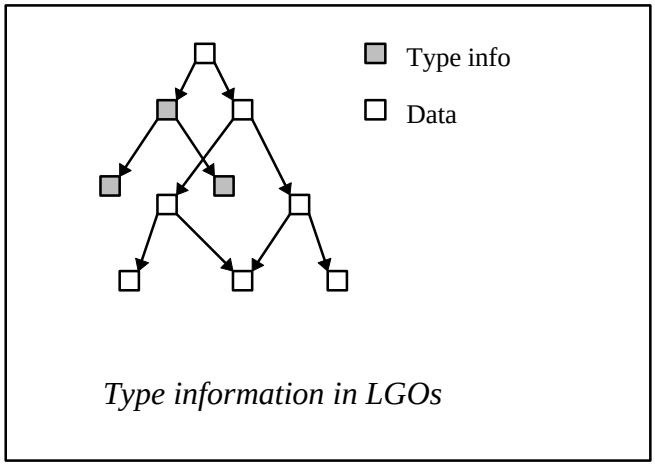
This is not considered a severe problem in Barbados, partly because such structures are actually quite rare, partly because there is a lot of stack space compared to the amount each recursion uses, and partly because it would be easy to write a more advanced algorithm in future versions which caters efficiently for all cases.

10.4.3 Type information

Type information is not stored with each dynamically allocated object. Instead, the saver discovers the type of each object by induction from the type of the root.

Each LGO stores sufficient type information to restructure the information contained in memory. This restructuring is accomplished as follows.

The root object consists of a value and a type. (It can be thought of as two pointers - one to the type and one to the value). The type information can be regarded as a complex data-structure in its own right: a class or structure often contains pointers to other classes/structs. The saver works by recursing down the tree/graph of type information first and then the actual data. In this way, all the type information needed for the data-structure is contained within the LGO along with the data itself.



Type information in LGOs

I now believe it would have been preferable to store a pointer to type information inside each heap block. A cleaner design would have resulted.

For instance, not having these type pointers complicates the persistence mechanism where inheritance is involved. Using inheritance, it is possible to have two pointers to the same object, where the pointers have different types (one type is a sub-type of the other). One attempt to get around this problem would involve always assuming an object is an instance of the largest class X where a pointer to X exists to the object. We might have a situation where all pointers to an object are sub-type pointers. However, this only matters where virtual functions exist, and this problem can be solved by checking the v-table[8] pointer inside the object.

To summarise, the problems of using induction to determine an object's type can be solved in this situation. However, this is an example of the problems that arise through separating dynamic objects from their types.

One disadvantage of having type pointers is that the user is then forced to use the 'new' operator rather than 'malloc()'. This is an issue for maintaining compatibility with existing C programs (and some C++ programs). However, there may be ways around this problem.

Another disadvantage of type pointers is that they consume 4 or 8 bytes in each heap block. However, many C++ objects require a pointer to a v-table anyway, and since a v-table pointer can be combined with a type pointer, there is no real saving from not having them.

More and more C++ compilers are beginning to support run-time type information anyway, because they see various advantages coming from it.

Finally, if the Barbados type-system is to be applied to languages other than C++, it would be advisable to store type information with heap objects. For example, the type systems of Java, Eiffel, and Python all require this information. Without this feature, the type systems would be incompatible. For this reason, future versions of Barbados will implement the storing of type-pointers in heap blocks.

10.5 Comparisons with other Systems

The SHORE system [Car94] implements large-grained objects which consist of a 'core value' (small object) and an associated 'heap' of other small objects. This concept is quite similar to Barbados LGOs. However, objects are accessed using calls to the database manager, and therefore this implementation would not suit Barbados' goal of providing for intensive computation on data-structures. Also, the C++ -based language implemented in SHORE has different persistence controls to Barbados. In Barbados, data-structures are explicitly moved into memory and back to disk, with two calls, whereas in SHORE an object is modified via 'transactions' that require each modification to be invoked by suffixing an object with the '->update()' member function.

SOS [Sha89a] had a similar object-model: there are 'objects' consisting of a 'primary segment' and arbitrary structures of 'indirect segments'. In SOS, these small objects are brought into memory upon demand, which is done transparently using virtual memory hardware. They are then written back to disk as main memory becomes scarce. In Barbados, 'LGO' I/O is user-level, i.e. the OpenLGO() and CloseLGO() calls are explicitly invoked, which I would argue has benefits for memory protection, locking etc.

[8] A v-table is a table of pointers to virtual functions pertaining to a particular class.

Also, persistence in SOS was implemented in C++ as a base class rather than orthogonally to type.

The IK system [Sou94] implements a clever form of orthogonal persistence based on 'clusters'. A cluster is a group of related small objects such that only one object (the 'head') is referenced from outside the cluster. References can be taken to any of the internal objects, although this will then cause the cluster to break up and that internal object will become the head of its own cluster. This system achieves the efficiency benefits of large-grained objects (intermediate structure) although it misses out on the semantic benefits.

Napier [Mor89] [Atk86] is a complete self-contained persistent programming environment. At the broadest level, it is similar to Barbados. However, it is different in the fact that memory protection is provided via type-safeness, and therefore type-unsafe languages such as C++ cannot be supported. Also, the type system is different - it is not possible to nest structs inside structs. This means that there will be more calls to the object creation routines. Furthermore, there is no concept of a large-grained object or intermediate-structure - something they would argue is desirable but which I have argued is not.

Grasshopper [Dea92] [Lin95] has a concept of a 'container' which acts like a large-grained object. While it claims to be an 'orthogonally persistent operating system', it has no support for fine-grained objects. Issues such as how to store data-structures and how to share them without having clashes in the address space have not yet been tackled. Instead, Grasshopper provides the infrastructure for persistent systems (such as Barbados) to be mapped onto it.

MONADS [Ros85] has coarse and fine-grained objects. In fact, [Hen93] specifically discusses these two levels of granularity. This system relies on specialised hardware and has fine-grained memory protection based on capabilities, which had its advantages but would be inappropriate for the data-structure intensive applications which Barbados was designed for.

ObjectStore [Lam91] is a commercial persistent object database system. Object accesses are managed through virtual memory techniques. However, it was designed with database applications in mind and does not have the necessary clustering ability to efficiently process complex data-structures.

The E++ system [Ric93] implemented a persistent version of C++, but this was designed more as a database programming language than a language for supporting intensive computation on data-structures.

10.5.1 Advantages of this form of Persistence

The following table summarises the advantages of the Barbados form of persistence.

Semantics:

a) Persistence is orthogonal to type.
b) All objects are automatically persistent until deleted.
c) The full C++ type-system is provided, including nested objects.
d) Memory protection is provided mainly by LGOs.

Efficiency:

e) Pointer dereferences are by straight machine-level dereferences.
f) LGOs are stored and transmitted in a compact form.
g) With a native-code compiler, it should be no slower than traditional UNIX/C systems. (?)

Flexibility with Data:

h) LGOs can go anywhere in an address space ("portable data-structures")
i) It is easy to examine multiple LGOs simultaneously.
j) Type-safe data can be mixed safely with type-unsafe data.

Intermediate Structure:

k) A standardised, enforced grouping for applications and system tools.
l) File-level operations.
m) The unit of i/o = unit of locking: Distribution is easy.

Some of these points require explanation. For example, benchmarks do not exist yet to support point (g). The assertion is made on the basis of an argument that a running Barbados application is similar enough to the corresponding UNIX/C application that the run-time should be about the same. Code that runs in-between LGO opens and closes should be virtually identical, and LGO opens and closes should occur with about the same frequency as file opens and closes since they are objects of about the same granularity and both contain natural 'packets' of logically related data. The author acknowledges that this is not quite satisfactory, and further development / benchmarking of realistic applications will be the next stage in the development of the Barbados Persistent System. A more serious qualification is the fact that the current version of Barbados runs inside a virtual machine (rather than in native code), which naturally carries a large cost.

Point (h) refers to the way the Saver and Loader move data-structures in-between address spaces. Under this system, it becomes easy to both reap the benefits of having convenient-sized 'packages' of data without suffering the consequence that it's hard to merge/cross-correlate information in different packages.

Point (j) is a consequence of the form of memory protection provided by LGOs. An LGO on disk is guaranteed to be type-consistent. Therefore, if a C program writes out a LGO, therefore, it can be then accessed by a type-safe language without losing type-security. Type errors can only occur when executing a program created by a type-unsafe language.

10.5.2 Disadvantages of this form of Persistence

The major criticism of this form of persistence is that it may make it harder to arrange data. In general, references cannot be expressed between fine-grained objects in different LGO's.

In addition, as programs evolve, it may happen that previous decisions regarding the arrangement of data within LGO's were not optimal. In this case, various modifications to code will be required and previous data will need to be converted.

10.6 Discussion Questions

10.6.1 Why doesn't the system support orthogonal persistence?

Barbados was specifically designed not to support orthogonal persistence. A fundamental premise of the system was that persistent stores should be organised around some kind of intermediate structure, i.e. small objects should be grouped into containers of some kind. The alternative is orthogonal persistence, where the persistent store is essentially a huge pool of small objects.

There is some confusion about the definition of orthogonal persistence and related concepts. Let us use the term '*type-orthogonal persistence*' to denote a style of persistence where objects of any type can persist. The alternatives to type-orthogonal persistence include systems where objects must belong in a parallel type-system in order to persist, e.g. the E language [Ric93], and systems where persistent objects must inherit from some 'persistence' base-class, e.g. Eiffel [Mey88].

'*Orthogonal Persistence*' will then be defined here to mean the stronger form of persistence, where the system implements type-orthogonal persistence but in addition provides the feature that any object can reference any other object in the system, and there is only one type of pointer. This arrangement is equivalent to having a 'single-level persistent store'.

Barbados does not provide orthogonal persistence, because it implements 'Large Grained Objects' with associated restrictions. This means both that there are two kinds of pointers, (namely ordinary memory pointers and LGO pointers), and that there is the restriction that fine-grained objects within one LGO cannot reference fine-grained objects within another LGO.

Proponents of orthogonal persistence, most notably researchers from the Napier project, argue that orthogonal persistence is desirable because it makes modelling data easier and removes the inevitable barrier that arises between large-grained objects. It simplifies the programming language and removes from the programmer the need to partition data into large-grained objects. Orthogonal persistence might also improve performance because the persistent system may be in a better position to partition and cluster data into physical locations than the programmer.

I believe that orthogonal persistence has associated costs, which outweigh the benefits. Orthogonal persistence does not scale well to large systems, for the following reasons.

Orthogonal persistence means that there are no longer any convenient 'packages' of data which the operating system and applications can use for transferring data, setting and checking permissions, specifying deletion, and for using in other higher-level operations such as version control and recompilation. The pattern of references between objects can become quite complex and begin to resemble 'spaghetti', so that the programmer no longer has a good high-level view of the data.

Orthogonal persistence makes it difficult to specify permissions and check permissions: either permissions work orthogonally to types, which makes it tedious to specify them and means that an object access can fail at any point in the program; or special 'permissions' objects are inserted as 'gateways' into objects, which means that measures must be taken to ensure that users cannot 'sneak' into data-structures from other paths. This latter alternative, in which permission checks are aligned with types by

being inserted into data-structures at special places, begins to resemble large-grained objects anyway, along with the associated restrictions and costs.

In a traditional programming language, files can only be modified by being opened with write-permissions, written into, and closed again. The same protocol applies to Barbados LGO's. However, in an orthogonally persistent system, there is no concept of a file or LGO. This means that it is possible for a bugged program to corrupt the entire persistent store, albeit in a type-safe way. For instance, a program could go into a directory near the root of persistence and remove a reference to a large body of data. If this means that the data is no longer reachable from the root of persistence, it means that the data is lost forever.

The benefits of an intermediate structure are discussed in [Coo95]. To briefly summarise this chapter, the idea is that organising small objects into containers allows certain operations to be 'factored out', i.e. done once for the group rather that once per small object. If these containers are system-level objects, then this allows the system to factor out system-level operations and hence be more efficient than otherwise. If these containers are user-level objects, then various user-level operations can be likewise factored out.

For example, when the user wants to transfer some structured data across a network, they inform the system which LGO or LGOs they wish transferred. The alternative is to select small objects individually or implement some 'transitive closure' rule (which makes it problematic to reference large objects e.g. compilers within user data.)

The paper [Coo96c] discusses the debate between orthogonal persistence and large-grained objects more directly.

To summarise my position, I believe that large-grained objects, whether they are 'files' or Barbados-style 'LGOs', are a natural part of computing. It is a useful discipline to force the programmer to arrange data-structures into higher-level objects, because this higher-level structure can be used by the persistent system or by programming tools or by the programmer themselves for various important purposes.

10.6.2 Isn't the Barbados concept of an LGO overloaded (i.e. it tries to do too much)?

In Barbados, the LGO performs many functions. Permissions are specified at the LGO level. LGO's can be moved, copied and deleted. LGO's are the units of migration between disk and RAM. LGO's are the units of sharing. LGO's can be used by applications for such things as version control and configuration management. Barbados already uses LGO's to factor out dependency information for program components.

Some would argue that each of these features is an independent function, and deserves an independent concept of 'large-grained object'. That is, the system should have many overlapping, orthogonal types of object groupings, for example 'protection domains' for permissions, 'packages' for transferring data and so on. Any object could belong in some of these groupings, but not others.

However, I believe that these groupings should always be aligned, e.g. with Barbados LGO's. There is enough overlap between the groupings that multiple orthogonal groupings would become confusing. For instance, when accessing a data-

structure, it is useful to have just one point at which permissions are checked, and at which the program can detect success or failure of the access. If the user has permission to access the data-structure, then the rest of the program can proceed. If memory protection (permissions) can be specified on arbitrary fine-grained objects, then the programmer will need to insert checks at many places in the program in order to check that the access succeeded.

10.6.3 Isn't Persistence supposed to remove the burden from the programmer of dealing with things like OpenLGO / CloseLGO calls?

Barbados programs which make full use of the Barbados persistence features will contain various calls to the OpenLGO() / CloseLGO() / CreateLGO() operators. The programmer will have to insert these calls at the appropriate points, and they will have to be aware (in some sense) of which objects reside in which LGO's. This is a disadvantage of a system such as Barbados when compared to a system providing orthogonal persistence. With orthogonal persistence, the user is not concerned with such issues.

I believe that the calls to OpenLGO() / CloseLGO() will not be very frequent in the source-code of any program. This is because LGO's are (by definition) large-grained objects. This means that a single call to OpenLGO() will make a large amount of data accessible. Many significant data-structures will be able to reside in a single LGO.

Similarly, it will not be too difficult to maintain an awareness of which fine-grained objects reside in which LGO's. As I anticipate Barbados being used, there will usually be relatively few LGO's open at any one time. In particular, there will be even fewer LGO's open for write-access, which is the situation where it is most important to know the precise location of objects.

10.6.4 How big should LGO's be?

In order for LGO's to work effectively, both in terms of system performance and in terms of programming, they must not be too large or too small. For example, if LGO's were frequently below 100 bytes in length, system performance would degrade through overly frequent disk accesses, and programs would have excessively many calls to the OpenLGO(), CloseLGO() and CreateLGO() functions. If LGO's were frequently above 1Mb in length, performance would degrade because too much information would be dragged in from disk and there would be pressure from users for programs to be re-written to provide finer-grained protection and to allow finer-grained sharing.

10.6.5 How do you know where to put the LGO's?

In Barbados, the user must organise data into LGO's. Many types of applications use medium-sized data-structures, such that each data-structure fits effectively into a single LGO. In this case, there is very little to decide when organising the data. However, larger data-structures need to be broken up into smaller pieces, each piece corresponding to an LGO.

Large data-structures are 'broken up' by replacing pointers in type-definitions with LGO pointers. Then, the dereferences of these pointers must be replaced by OpenLGO() / CloseLGO() calls.

Proponents of orthogonal persistence may argue that this makes it difficult to translate real-world concepts into large and fine-grained objects. For instance, the programmer may not be in a position to know when writing a program what the best way is of dividing data into LGO's. Or worse, the pattern of usage of a program might change over time, so that the decisions made at the start about where the borders between LGO's should go, will become suboptimal.

I believe that most data structures do naturally fall into 'clusters' of objects. The programmer can easily pick some logical level at which objects are grouped in easy-to-handle sizes. For example, in a word-processor, the 'document' or 'chapter' would be chosen as the unit corresponding to an LGO. In a CAD-CAM system, some component sub-assembly would be an LGO.

If the programmer picks some partitioning which is suboptimal, e.g. they write a program such that an LGO corresponds to a 'section' instead of a 'chapter', it will often not matter much. LGO's can still cope efficiently with a large range of sizes. If the programmer's partitioning really is inappropriate, and the program needs to be rewritten e.g. to support finer-grained sharing of data between users, I believe the changes to the program are significant but not so large to threaten the paradigm. The old program and the new program still use the same types, and have the same function calls. The changes which need to be made are to change LGO pointers to ordinary pointers or vice versa, and to change OpenLGO() calls to pointer derefences or vice versa. The most difficult aspect of a reorganisation of this kind is inserting OpenLGO() / CloseLGO() pairs of functions into the appropriate parts of a program.

10.6.6 Can't containers or large-grained objects be built on top of an orthogonally persistent system?

Obviously, an orthogonally persistent system can be programmed in such a way that entities corresponding to 'large-grained objects' are provided, by building them on top of the fine-grained objects. However, such a solution will contain all the disadvantages of large-grained objects (e.g. restrictions on what objects can reference what objects), and it will furthermore have the performance disadvantage of not implementing large-grained objects as system primitives.

In order to attain all the benefits of large-grained objects, there should be *standardised* and *enforced* rules governing the organisation of fine-grained objects in large-grained objects.

Enforcing large-grained objects:

For example, there should be mechanisms preventing programmers from violating large-grained object borders (having fine-grained references which cross a large-grained object border), so that various applications can assume that such borders are intact. An example of such an application is the Barbados 'Make' program. This system-level program uses large-grained objects to cut down on dependency information and dependency checks, and it would not work effectively if it could not assume that large-grained objects are self-contained in terms of fine-grained references.

Standardising on large-grained objects:

A disk-usage utility is an example of a utility which can benefit from having a concept of a large-grained object. It can report on which objects are consuming the

most disk space. Such a function would not work at a fine-grain, because the fine-grained objects would be too numerous to provide sensible information. And in order to work with arbitrary data, the disk-usage utility must be aware of where the large-grained object borders exist inside arbitrary data-structures.

10.7 Summary

Barbados was always intended to be more ‘pragmatic’ than ‘ideologically sound’. Hence orthogonal persistence and type-safeness were passed over in favour of LGOs and more traditional forms of memory protection.

The resulting implementation of persistence is efficient, type-orthogonal, requires little change to languages or existing programs, and supports distribution.

Discussion of Barbados

11. Initial Experiences of Barbados

The Barbados persistent store is steadily growing in size. It currently includes a small program to display and rotate wire-frame pictures of 3D solids, a small class library, and a large application called 'tt'.

With the exception of 'tt', all the code was developed within the Barbados environment, using the Barbados editor and Barbados compiler and the Barbados persistence facilities. The only copy of the source-code for the programs (apart from backup 'logs') resides in the Barbados store.

The Barbados compiler compiles to a stack-based pseudo-code. The interpreter for this pseudo-code generally seems to 25 times slower than equivalent code produced by a native compiler, although there are opportunities for considerably optimising this (since it is a very low-level pseudo-code).

11.1 The 'tt' Program

'tt' is a high-school timetable construction system. It consists of 21000 lines of code, in 38 modules (directories), originally written in C under UNIX. It was ported to Barbados with almost no modification to the code.

The use of persistence in the 'tt' program is revealing. 'tt' consists of highly complex data-structures. The artificial intelligence techniques used inside it include network flow, bipartite matching, graph colouring etc. There are 500 lines of type-definitions alone. Since it is a C program, this means 500 lines of structure definitions and enum definitions.

11.1.1 'tt' version 1

In order to get 'tt' working in Barbados in the same way it worked in UNIX, very few modifications were required. It was necessary to arrange it into Barbados directories, and to write a couple of stub functions which are provided by the ANSI standard C library but not yet by Barbados. To port a C module in tt into a Barbados directory required replacing the "#include"s with directory references (i.e. executing declarations of the form "directory &X=../X") and compiling the whole source-code file.

11.1.2 'tt' version 2

However, this basic version of 'tt' did not store the data-structures persistently. The issue was dynamic arrays. In the code, extensive use was made of pointer vectors. These were malloc'd blocks consisting of lists of pointers. It was the primary method used to store collections of objects. However, this clashed with the persistence mechanism of Barbados, which requires the use of Barbados 'dynamic arrays' (i.e. objects declared with the '[?]' syntax) for such purposes. The persistent system was not aware of the type of these malloc'd blocks. It could infer that the first element in the block was a pointer of a certain type, but the remaining pointers in the block were not examined and hence were not swizzled.

To fix this, the type-definitions were modified so that the variables and data members became dynamic arrays instead of pointers. Furthermore, the code to allocate space and increase space for these blocks was commented out. No code was needed in its place, since the Barbados dynamic arrays automatically re-size as higher and higher elements are accessed. The result was a version of 'tt' which did operate on persistent data.

11.1.3 'tt' version 3

However, version 2 of 'tt' did still not qualify as a genuine persistent program. This is because the data-structures it operated on were hard-wired into the program, which means for instance that it would be difficult to manipulate data-structures for different instances.

'tt' is a program to generate high-school timetables. It operates on only one school at a time. However, associated with one school are several data-structures, each one reachable from some 'root' global variable in the program. These global variables are components of the program. They reside in the directory corresponding to the global 'typedefs' header file. Let us call this directory "/tt/tt", and let us assume that it corresponds to an LGO. Therefore, the model of persistence in 'tt' version 2 was to have all the data-structures resident in the '/tt/tt' directory. Also, for the 'malloc()' calls to work properly, it was necessary to either start the program from inside the "/tt/tt" directory or set the default heap to the "/tt/tt" directory. Furthermore, to apply the program to a different school would involve losing all the persistent data from the current school.

The appropriate organisation of data in this application would be to define a new class, called a 'Timetable', which contains all the roots of the various data-structures. Then, an object of this type could be manipulated by the program. Multiple LGOs could then exist, each of type 'Timetable'.

Such a structure would be natural in an object-oriented language such as C++. Each of these data-structures could be processed simply by invoking member functions on them. However, because the program was written in C, and because it was written without persistence in mind, there was no encouragement to organise the data in this way.

Version 3 of 'tt', which has not yet been written, would involve creating a class 'Timetable'. The simplest method of getting the program to work with this class would be to write member functions to 'link' and 'unlink' instances of 'Timetable' from these global variables. The vast majority of 'tt' would then not need rewriting.

11.2 The 'Spin' Program

This 'Spin' program was a program developed completely within the Barbados paradigm. The program displays wire-frame solids rotating in 3 dimensional space.

'Spin' was developed from the bottom up. Various base classes were developed, starting with the simplest classes (3-D vectors, 2-D vectors, bitmaps) and building up to the more complex classes (a 'camera' class, a 'solid' class, and a 'scene' class which corresponds to a (camera, solid) pair).

Each class was tested before the later classes were built on top of them. The vector classes were tested by checking that the output of the various operators (dot

product, cross product etc) corresponded to paper calculations. Then 3-D lines were displayed inside bitmaps. Then solids were displayed in 3 dimensions. The final step, to make the solids spin, was very easy.

There was a large jump in going from a newly-written ‘camera’ class which can only display single lines in the bitmap, to a ‘scene’ class and a ‘solid’ class which allow full 3-D wire-frame polyhedra to be displayed. There did not seem to be any easy intermediate way of testing that the ‘solid’ class was functioning correctly. Fortunately, the code did not require much debugging.

This program was written before debugging was implemented in Barbados, and even now debugging is not fully implemented. However, being able to interactively execute functions and query data was almost as good as having a full debugger.

The program was developed without using existing code. The classes did need to be modified many times between the first compiled versions and the final program, but at no stage did this result in a noticeable pause for recompilation.

12. A Comparison of Barbados and Napier

The system most similar to Barbados is Napier. Napier is a type-safe, orthogonally persistent language and system developed primarily at the University of St Andrews and the University of Glasgow. Napier is quite a stable product with a long history and a large amount of literature associated with it. The first major release of Napier was in 1988.

Napier was designed with an uncompromising attitude that complete orthogonal persistence should be provided along with strict type-safeness. By contrast, Barbados was designed with a more pragmatic attitude, so that useability issues and performance issues would be considered, even when at the expense of a pure user-model.

12.1 Orthogonal Persistence

Orthogonal Persistence means that data is accessed in a uniform manner, regardless of its creator, longevity or type ([Atk83]).

Napier provides orthogonal persistence. Barbados does not, because of the restrictions introduced by large-grained objects: a fine-grained object in one large-grained object is not allowed to contain a reference to a fine-grained object in another large-grained object (with a couple of exceptions - see Appendix B: Persistence). However, Barbados does implement 'type-orthogonal persistence', meaning that objects of any type can persist.

I do not consider it desirable to implement full orthogonal persistence in the Napier sense. This position is argued in [Coo95], [Coo96c], [Buh89], Appendix B and see also [Hen93]. I believe that the concept of a large-grained objects is something intrinsic to programming. The issue is discussed at length in Appendix B.

12.2 Type-Safeness

Type-safeness means that by providing strict typing rules, the programmer is prevented from committing memory errors. Memory errors include overwriting array bounds, corrupting the heap, and writing outside allocated areas of memory.

The most insidious and time-consuming bugs that occur in C programs (and programs in other languages) are to do with memory errors. Memory errors are caused by (a) overwriting the end of an array, (b) using the type-cast operator to create pointers out of non-pointer values/pointer values of a different type, or (c) using 'union' types to create invalid pointer values.

They are especially insidious because the effects can often be hard to relate to the causes, and they can interact with the debugging mechanisms, making it difficult to pin-point the problem.

In a persistent system, it is especially important to have adequate memory protection, because in a persistent system 'memory' can refer to a computer's entire file-system.

I chose not to implement type-safeness because I saw it as an important goal to be able to support a wide range of programming languages, many of which may not be

type-safe, and I wished to support C++ in particular because of its popularity. (By supporting a popular language, source-code for real applications was readily available.)

Barbados relies on memory protection methods other than type-safeness. The primary method of memory protection is provided by LGO's. Because Napier wishes to support orthogonal persistence, they do not have this form of memory protection available, and hence they rely solely on type-safeness. (Actually, Napier has something similar to protected LGO's in the form of 'hyperworlds'. However, these 'hyperworlds' are still so large that finer-grained memory protection is required.)

One of the additional advantages of not supporting type-safeness is that programmers are able to implement efficient low-level applications such as compilers and data-compression utilities. In Napier, if the programmer wishes to write a compiler for a language other than Napier, they must either transform the source-code into Napier source-code and pass it through the Napier compiler, or they must produce their own virtual machine-code to be interpreted by a Napier program. To implement data-compression utilities can be difficult because type-safeness prevents pointers being 'compressed' and 'reconstructed'. However, at the same time it is acknowledged that (a) the majority of programmers do not write such low-level tools, and (b) provisions can be made to allow programmers to 'break' the type-system in exceptional cases.

Because I imagine memory bugs still being a thorn in the side of C/C++ programmers, I envisage Barbados having extensive support for debugging memory bugs. At present, in fact, Barbados checks the validity of pointers that are passed into 'free()' and 'realloc()' calls. It also has a 'debug mode' whereby each individual word of memory is tagged as being 'accessible' or 'not accessible', and each memory write in the virtual machine is checked against these tags. This system uses fractal techniques. These techniques are discussed in Appendix B: Persistence.

Nevertheless, there are advantages and disadvantages of type-safeness, and only extensive experience with IPPE's of both varieties will provide a conclusive answer to the question of whether an IPPE should be type-safe or not. I feel more strongly about orthogonal persistence than about type-safeness, and a type-safe version of Barbados is certainly conceivable. A Java-based version of Barbados would be such a system.

Appendix B ("Persistence") describes how it might be possible in Barbados to safely mix data from type-safe languages with data from non type-safe languages.

12.3 Referential Integrity and Garbage Collection

This section is more about persistence issues than programming environment issues, although the persistence model does impact the programming environment to some extent.

Referential Integrity means that "once a reference to an object in the persistent environment has been established, the object will remain accessible via that reference for as long as the reference exists", and furthermore that objects have one and only one (unique) identifier [Atk95].

Again, the Napier system has this property, whereas Barbados does not.

Garbage Collection is a process whereby the system detects objects not reachable from some 'root' reference or references, in order to reclaim the space used

by them. It is used as an alternative to explicit deletion. In a tree-structured network of objects, the effect of garbage collection can be achieved by reference-counting: when an object's reference count reaches zero, it can be removed. This is the system used in UNIX, which has a strict hierarchical structure of files. However, in a graph-structured network of objects, such as the Napier persistent store, it is possible to have cycles of objects which reference each other but which are not reachable from any root reference. Therefore it is necessary to perform full garbage-collection.

Referential Integrity is essentially about garbage collection. In order to provide referential integrity, it must not be possible to explicitly delete objects: only references can be deleted. An object must be deleted by the garbage-collector only after the last remaining reference to it is gone.

Barbados does not have garbage collection, or at least it only has it in a very restricted sense. Fine-grained objects are deleted using 'free()' or 'delete', and large-grained objects are deleted using the 'DeleteLGO()' function. The limited form of garbage-collection which exists is explained in Appendix B, however very briefly it just means that when a LGO is 'closed', garbage collection is performed on the contents before writing the LGO to disk.

The reasons for the decision not to provide garbage collection were:

(a) A philosophical belief that users should be allowed to delete data when they want to, either for privacy reasons or because the data consumes a lot of memory, (NB: this applies mainly to LGO's, not fine-grained objects)

(b) There are costs associated with garbage collection, namely that garbage fills up RAM which could otherwise be used to cache data, and that garbage collection which is run automatically introduces an awkward pause in running programs, and that often in systems with garbage collection, data is not clustered on disk for efficient access until the global garbage-collector has run.

(c) There was also the desire to support the C++ language. The C++ language does not have garbage collection, and efficient programs can be written as a result: if hundreds of objects are being created during the running of the program, and the programmer can inform the system of which ones are no longer needed, then the system can immediately re-use the memory occupied by them and hence use memory more efficiently and save on the cost of garbage collection.

(d) There did not seem to be great advantages arising from the concept of referential integrity, at least within the Barbados paradigm.

However, a lot of research has gone into improving garbage-collection algorithms, so that they can run in the background and so that they can run across a distributed system [Sha92] [Pla92]. Depending on the effectiveness of these algorithms, points (b) and (c) may not be valid.

Regarding (d), in [Mor93] Morrison et al argue that referential integrity can be used to build better programming environments. However, these arguments are more to do with pointers being used instead of symbolic names to manage the various associations between procedures, source-code, version information etc.; a feature which Barbados does support.

12.4 Graphical vs Text Programming Environments

Research on integrated programming environments in Napier has focussed on the construction of a 'hyper-programming system' [Kir92] [Mor95]. In this system, a graphical browser is used extensively to identify objects and execute code.

By comparison, Barbados is a command-line based programming environment. The mouse is rarely used, and little use is made of windows.

I believe that a text-centred interface of this kind, where the user seldom needs to use the mouse, and where output is interleaved with commands, is the appropriate programming interface for the majority of programming tasks and the majority of programmers. I believe that such an interface provides more throughput between human and computer. This opinion is based mainly on my own experiences of GUI systems but also on comments made by others. A mouse can be a slow way of issuing commands and identifying objects.

However, it is conceivable that a Napier-style GUI be implemented in Barbados. The Barbados graphical browser is a step along this direction. Similarly, a Barbados-style command-line could be implemented within Napier.

12.5 Text programs vs Hyperprograms

In a hyperprogramming system, source-code actually contains direct links (similar to hyper-text links) to other program components. These links are inserted by a graphical browser as the user writes the code.

Barbados does not support this representation of source-code. Compiled code objects contain links to other objects but source-code objects are plain text.

This means that Barbados does not have the advantages of hyperprogram links, for example the wide range of linking times and being able to represent source-code independently of the context (i.e. name-space, or directory) in which the function resides.

Supporting hyperprogramming features was not a high priority, partly because I don't believe the advantages are great, and partly because I believe that programming should be primarily a text-and-keyboard activity. I believe that many naming/binding issues should be solved at the configuration management level, dealing with large pieces of programs, rather than at the fine-grained programming level.

However, the decision not to support hyper-programming may just reflect personal biases. It is conceivable that hyper-programs be implemented in Barbados. The Barbados compiler already has an internal pseudo-token which is used to symbolise a resolved path-name, and this mechanism is very similar to the mechanism which would be required to compile hyper-program links.

It should be noted that a flat representation of programs makes configuration management issues (e.g. version control) straightforward, whereas a hyperprogram program representation may cause difficulties for a version control system. However, both Barbados programs and Napier hyper-programs suffer problems of this sort: a version control system for any IPPE would have to be able to cope with the structure of IPPE programs, where programs are organised as structures of many small components.

12.6 Object-oriented / object-based vs non-OO languages

An object-oriented language is defined by [Weg89] as a language with the following properties:

1. The ability to define an object as a set of operations and a state that remembers,
2. The objects can be categorised by class (type),
3. There is an inheritance mechanism for defining superclasses and subclasses.

A persistent system inherently focuses on the concept of a 'typed object', and it would seem natural for the system to have the advantages of object-orientation as well as persistence.

Barbados implements a language very similar to C++. Classes, member-functions and multiple inheritance have all been implemented, and hence Barbados can be called an object-oriented system. (Virtual functions are the next remaining feature to implement).

Napier is not an object-based language. Napier objects can have member-objects which refer to functions, however these pointers are not automatically initialised.

While object-orientation is not central to the design of Barbados, object-orientation and object-basedness can yield many advantages to an IPPE:

- Object-orientation is recognised as providing many advantages to general programming and software engineering,
- Object-basedness simplifies naming/binding issues: once an object has been declared as having a certain type, the majority of the functions which apply to that object can be found directly as member functions. This means that
    - You can list the available operations on an object with a command such as 'Help()' in Barbados,
    - You do not need to specify path-names to the methods, they are automatically type-checked and linked with your code.
- It is (arguably) easier to remember how to use the Barbados meta-level *classes* than to remember a how to use a set of meta-level *functions*. For example, to convert a 'named_obj' to an 'any' requires a simple cast.
- With object-basedness, values can be displayed automatically on the log by having a 'Print()' member function invoked on them. In a functional system, some naming convention would probably be required instead. Also, inheritance can save the user from implementing a 'Print' method in every class.

What Napier has instead is the concept of an 'abstract type'. An instance of an abstract type is a record containing data fields and other fields which store function references. The instance is initialised by passing the various function pointers as arguments to an initialisation operator. If this construction process were wrapped up in a function, one would have something approaching an object-oriented language. Similarly, in order to achieve inheritance, the programmer is required to do some work for each sub-class.

In Napier, the functions which pertain to a particular class have references inside each instance of the class. In C++, these function references are factored out into an object called a 'v-table' (virtual function table). Use of 'v-tables' yields a more efficient

implementation of object orientedness and encourages programmers to write large numbers of member functions per class.

Another relevant issue is: "what happens to existing instances as the class methods evolve around them?" In Napier, each existing instance will remain bound to the old versions of the functions (static scoping). In Barbados, the existing instances will be automatically bound to the new versions (dynamic scoping). There are situations where either system is preferable. Experiences of Barbados are not sufficient for a definitive statement to be made as to which is better; however it should be noted that with programs under development, existing class instances do not play an important role (hence it is not an important issue). Note also that as software becomes more stable and 'releases' are made, existing class instances become more important, however supporting them becomes an issue of configuration management: something which is discussed briefly in chapter 14 but essentially not dealt with in this thesis.

12.7 Reflective Programming

Reflective programming refers to the practice of having one program create or modify another program, or even itself. A typical way of doing this is to construct a body of source-code which is then passed through a compiler.

An IPPE provides much better facilities for reflective programming than traditional systems, because the compiler can be called from within the store, and it can be invoked on relatively small amounts of source-code, and the resultant procedures can be executed from within a running program.

In particular, Barbados provides support for reflective program by (a) the provision of a compiler that can be called from within the system, namely the "compile()" function, and (b) the meta-level classes, which among other things allow the user to execute a procedure which has been created or identified at run-time. (The 'FunctionCall()' member function of the class 'any' allows this).

Reflective programming has not yet been experimented with in Barbados.

A large amount of research on reflective programming has been done in the Napier project [Kir92b] [Kir93].

12.8 The Meta-level

The Barbados meta-level classes provide the user with the means to manipulate types, directories and dynamically-typed objects. They do so mostly from within the C++ language.

By contrast, in Napier the language has been designed to provide these facilities. For example, special syntax exists which allows programs to create new directory entries (environment entries), remove entries from the directories, to create values of type 'any' and to project values of type 'any' onto specific types.

The Barbados approach leads to more verbose syntax, but more flexibility. For example, consider the following examples:

Task	**Barbados**	**Napier**
Creating a new object 'foo' in directory D:	`D.Create("foo", typeof(int));` `= foo;`	`in D let foo = 3`

directory D:		
Non-interactively creating and initialising 'foo':	`named_obj foo=D.Create( "foo", typeof(int)); (int)foo.Any() = 3; = 3`	`in D let foo = 3`
Deleting an object:	`D.Delete("foo");`	`drop foo from D`
Deleting a user-specified object:	`cin >> objname; D.Delete(objname);`	`! Compile & execute the string ! "drop "++objname++" from D"`

12.9 Browsers

A 'Browser' in the context of persistence refers to a program which allows a user to interactively examine and explore a persistent store, and interact with the data inside.

For example, they could begin at the root of persistence. Then, using a mouse, they can click on successive references. Each reference would then open up the object referenced, revealing more links to other objects. Functions can be called in this way, and values can be modified.

A sophisticated browser has been developed for the Napier system with these features.

Barbados contains a program called 'Browse' which is an embryonic form of a browser for a persistent system. This program can take an arbitrary object, and display it and its transitive closure (i.e. other objects reached directly or indirectly from it) as a graph in 2 dimensions in a reasonably optimal layout. However, it lacks a great many other features a persistent browser should have.

While the Napier browser does not have the graph visualisation feature, it has a great many more features. These include being able to interact with objects to select certain objects to be opened up, to modify their fields, and to call functions which are found from the browser.

12.10 Summary

Napier is a language explicitly designed to support persistent programming, especially research into persistent programming. It is a small language, designed for maximum orthogonality of features.

In the design of Barbados, more importance was given to useability and efficiency issues.

Napier is a type-safe language, which carries obvious benefits but which places restrictions on lower-level forms of programming and makes the system incompatible with non type-safe languages such as C++ and Pascal. Barbados is not type-safe, which can lead to more bugs but gives greater freedom to programmers and makes the system compatible with a greater range of languages.

Napier is a garbage-collected system, which means that programmers are not concerned with storage issues. Barbados is not a garbage-collected system, which means programmers must think about storage issues.

Barbados has the concept of a ‘large-grained object’, which leads to a more complex model of persistence but which yields various advantages.

13. Comparison of Barbados with Smalltalk

After Napier, the system which is most similar to Barbados is Smalltalk.

13.1 The Design Philosophy

Smalltalk [Lal90] was designed to be a pure object-oriented language, and to support large-scale software engineering, especially large business applications. It also supports Graphical User Interface (GUI) programming quite well, with a large class library for graphics and windows. Even the design of the language was influenced by the desire to support GUI programming: there is a message-passing rather than function-call protocol.

Everything that can be manipulated in Smalltalk is theoretically an object, in fact even classes are objects. Class instantiation is provided by sending a 'new' message to a class object and waiting for a reply. The language is dynamically typed. Programs go through a parsing stage and the resultant syntax trees are interpreted.

Both Smalltalk and Barbados claim to be better than traditional compiled environments because they support incremental and interactive programming. Barbados has the additional advantage that it generates compiled code (or at least is designed to generate native code) and hence produces faster programs.

The Smalltalk language is probably easier for beginners to learn, whereas C++ is ultimately more powerful. In this sense, they each cater for slightly different users: average programmers will usually prefer Smalltalk, and advanced programmers will usually prefer C++.

Both systems aim to assist programmers by providing a complete, integrated programming environment.

13.2 The Programming Experience

Smalltalk claims that programmers program by 'extending the class library'. Smalltalk programmers begin at a 'class browser'. A 'class browser' is a window which allows users to browse through the class library and through the methods for each class, and view and modify the code that implements each method. A workspace can hold multiple class-browsers.

The following is an example of a Smalltalk class browser:

System Browser

Graphics-Prim	----------	accessing	hilberts:
Graphics-Disp	Pen	coloring	mandala:diame
Graphics-Path	Point	moving	spiral:angle:
Graphics-View	Instanc|Class	geometric des	----------

```
spiral:n angle:a
      "Draw a double spiral directly on the display."

      "Display white.
      Pen new spiral: 200 angle: 89; home; spiral: 200 angle: -89."

       1 to: n do: [:|self go:|; turn: a]
```

In Barbados, classes can occur anywhere in the directory hierarchy. In Smalltalk, classes are stored in a centralised set in a two-level hierarchy. This two-level hierarchy consists of 'class categories' and 'classes'.

Class browsers come in two halves, with the top half being split into 4 panes. The first pane in the top row is the 'class category' menu. The second pane is the class. The user finds existing classes by selecting items from these two windows. Class methods are likewise organised into a two-level hierarchy of 'message categories' and 'messages'. Panes 3 and 4 allow the user to select individual messages, i.e. methods.

Once a class has been selected, it can be viewed or modified in the lower window. Alternatively, a 'message' can be selected and likewise viewed or modified.

The user can execute or evaluate program fragments by highlighting source text with the mouse and selecting a menu option to 'do it' or 'print it'. The source-text can be any text inside an existing 'message' or temporary text the user types in the bottom half of the browser. When objects are printed, they are printed using the 'printOut' message of that object.

Compared with Barbados, the Smalltalk environment is more dependent on the mouse (or pointing device). However, it could also be said to provide a more user-friendly interface to the user. Barbados is probably more suitable for experienced users.

13.3 Programming Tools

Smalltalk and Barbados are both similar in that they are both highly integrated environments yet at the same time quite 'open' systems. They are 'open' in the sense that many of the programming tools are implemented by ordinary objects which exist in the user's 'space', and which have the same status as the user's own objects. The alternative is an environment which provides functionality through built-in tools, accessed through pull-down menus and so on, and which can not be replaced or re-implemented or built upon.

An example is the functionality to search for objects which reference a given object. In Barbados, this is accomplished through the 'grepdepend()' function which is implemented using the meta-level classes, (including the function: `named_obj ::Dependency()` which is implemented at system-level). In Smalltalk, this is accomplished through various objects and messages such as the following:

```
Smalltalk browseAllCallsOn: (Smalltalk associationAt: #Display)
```

The 'browseAllCallsOn' message creates a browser. The set of objects that will appear in the browser is given by the subsequent expression, which in this case is a sub-expression which searches for references to the symbol 'Display' from the system library.

Smalltalk provides support for debugging, via various windows which allow the user to inspect object values, set breakpoints and so on. Because Smalltalk is more interpreted than Barbados, it is easier in Smalltalk to modify a running program. In Barbados, (at present) one has to return to the command-line before executing a modified program. However, because Barbados performs more checking at compile-

time, being able to modify running programs is not as important in Barbados as it is in Smalltalk.

13.4 The Language

Both Smalltalk and C++ are object-oriented languages, however C++ also supports function-based programming.

They both support multiple inheritance. However, while Smalltalk supports multiple inheritance, it does not occur in the standard library.

Both languages support information hiding. In C++, this is provided by the 'public' and 'private' keyword in class definitions. In Smalltalk, the public interface to an object is defined as the set of messages defined for the class.

Both languages support polymorphism: in C++, it is possible to have multiple functions with the same name, but taking parameters of different types. In Smalltalk, all objects are dynamically typed and hence methods are not chosen until run-time, so polymorphism is something intrinsic to the paradigm. Interestingly, this feature is used to implement boolean values in a unusual way: 'true' and 'false' are objects of different types, which both inherit from a superclass called 'boolean'. The various boolean operations are implemented as messages on the classes 'true' and 'false'; such that the messaging code for 'true' is different to that of 'false'.

13.5 Directories and Dictionaries

The Barbados notion of a 'directory' corresponds to the Smalltalk notion of a 'dictionary'. Both are collections of named, typed objects, and both can be nested. In practice, Smalltalk dictionaries are generally large and are not nested deeply. They are used more as 'workspaces' or 'contexts' than as a way of giving data structure. Path-names are not integrated into the language. Instead, the user needs to make a series of dictionary lookup operations, albeit with quite a concise syntax, however this leads to run-time resolution of paths rather than compile-time resolution.

13.6 Debugging

Both systems are designed to have extensive support for debugging. (Although in Barbados, only a very primitive version of the debugger exists to date).

Smalltalk supports debugging using windows which allow the user to inspect and modify values. Breakpoints are set by inserting a call to the debugger ('`self halt`') at the appropriate point in the code, and compiling the message. Barbados uses a different technique for implementing source-level breakpoints, (special virtual-machine instructions), but it can (and probably will eventually) use this technique. This technique of compiling debugger calls into code is only available to incremental programming environments, because of the cost of recompilation. Compared with the alternatives, this implementation might make it easier to integrate the debugger with local variables.

13.7 Summary

Both systems support incremental and interactive programming. They are both highly integrated, and yet ‘open’ systems. Barbados supports the C++ language, which is a rich and concise and efficient language. The Smalltalk environment supports the Smalltalk language, which is a small, concise and less efficient language. Barbados is a highly text and keyboard-based environment, designed for efficient use by expert programmers, whereas Smalltalk makes more use of windows and mouses and is designed for user-friendly use by less expert programmers.

14. Directions for Further Research

14.1 Parallel Programming

The concept of a LGO is something which could be useful in the domain of parallel processing. It allows data to be wrapped up into a single (flat) object and then communicated between different address spaces. These objects consist of logically related items, which display good locality of reference, and they come with a locking protocol, which is appropriate because they are logical entities (they consist of logically related items).

This system encourages large-grained concurrency. A LGO containing a program can be sent off, along with LGOs containing the parameters to that function, to another processor for processing. Global data would consist of the set of LGOs in the system - LGOs would be passed around by processors following a multiple-reader/single-writer protocol. LGO identifiers would be the persistent identifiers in the global memory.

A parallel program could consist of a number of parallel processes, each farming off sub-processes. To spawn a sub-process, the parent process would construct a 'function call', meaning a function reference and a set of parameters. The parameters would need to be meaningful outside the parent process's address space, and therefore should only contain non-pointer values and LGO pointer values.

14.2 Data-compression with LGOs

An 'LGO' is (Large-Grained Object) is a Barbados concept used in the implementation of persistence. Large-grained objects replace 'files' in many ways. they are described in appendix B.

Since the only operations on LGOs are to open and close them, and random-access is not supported, there is the possibility of providing data-compression at the operating-system (i.e. persistent system) level.

This optimisation would have the advantage (a) saving on disk space, which is a minor benefit, and *possibly* (b) saving time by doing fewer disk reads or reducing the volume of network traffic.

Whether it actually improves performance depends on many factors, such as the granularity of objects accessed, the compressibility of the data and the speed of the compression algorithm. It seems that fairly crude compression must be used if the goal is to improve performance.

14.3 Native Code Generation

At present, Barbados generates and then interprets pseudo-code. In order to get a proper idea of performance as it will be in the final system, it will need to generate native code.

Consideration will be given as to whether “Java Virtual Machine” to “native code” converters can be used instead of traditional code generation techniques.

14.4 Configuration Management

Configuration management is the broad problem which includes version control, building applications and supporting different platforms.

While configuration management is outside the scope of this thesis, some thought has been given to it.

A prominent part of the plans for configuration management support is a utility called ‘Release()’. Release() will take a specified large-grained object or objects, and wrap it up along with other large-grained objects it has code dependencies on, into a single (new) large-grained object called a ‘release object’. The links with source-code will be severed at this point, meaning that the program can evolve independently from the release object.

15. Conclusions

The aim of this thesis was to explain the concept of an Integrated Persistent Programming Environment (IPPE), why it is useful, and to describe the author's experience in building one.

An IPPE is a programming environment built on top of a persistent system. This means that source-code, executable code and data all co-exist in the persistent store. An IPPE is both an application of persistence as well as an interface to persistent programming. It can be used either as a programming environment on top of an otherwise normal operating system, or as a programming environment giving access at a low level to a persistent system.

The benefits of an IPPE in general, and Barbados in particular, include:

- Incremental compilation: this means that if a small change is made to a program, only a small amount of code needs regeneration.
- An interactive environment: this means that the user can run code-fragments and execute commands defined by their program (e.g. testbed routines), interactively.
- A monolingual environment: the same language is used as both a shell language and as a high-level language.
- There are no more makefiles: the environment automatically and transparently tracks dependencies and keeps programs consistent (both internally consistent and consistent with source-code).
- Visual feedback: the user can query data-structures, using print methods defined by their programs, with a minimum of typing. Results are printed underneath the corresponding command, which also aids feedback and programmer concentration. The results can be displayed in text or with bitmaps.
- A data-structure browser: for debugging or for other purposes, the user can use a graphical tool to inspect various objects and their relationships to each other.
- A number of tools exist which understand the user's programs to a fair amount of detail and hence can provide more intelligent services. An example is the 'grepdepend()' dependency searcher.
- Meta-level classes allow the user to define their own programming tools. These tools can be either general tools or project-specific tools.
- The environment encourages the user to write and use a large set of small tools, e.g. as in UNIX.
- Data exists alongside program components in the persistent store. Therefore the distinction between the two can be blurred for more flexibility. For example, GUI resources can be constructed using graphical tools, and then the resulting objects (e.g. icons) can be referenced directly by the program with an identifier.
- Debugging can occur in the same language, with the same compiler and in the same context (e.g. with C++ macros) as the code being debugged.
- The environment works with a very standard version of C++, and with quite standard compilation techniques (so e.g. code can be generated with the usual optimisations).
- The environment provides *persistence*, which reduces program size and complexity and provides opportunities for more advanced features and promotes sharing of data between applications.

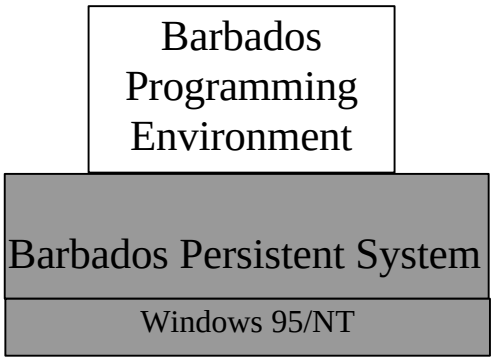
The author has completely implemented an IPPE, called ‘Barbados’. Barbados is significantly different from the only other IPPE the author is aware of, namely Napier.

The issues that arose in its construction included: what model of persistence is appropriate for an IPPE, what an effective user-interface is for an IPPE, how fine-grained program components can be kept up-to-date with each other, how a programming language needs to be modified for an IPPE, what new programming tools arise in the context of an IPPE, and how meta-level classes should be designed to allow the programmer to interact with types, named-objects and directories.

In conclusion, sufficient research into IPPE’s now exists that an IPPE can be commercially developed.

16. Appendix A: Implementation of the Barbados IPPE

This chapter describes the implementation of the programming environment aspects of Barbados. As the following figure illustrates, the Barbados programming environment is supported by the Barbados Persistent System.



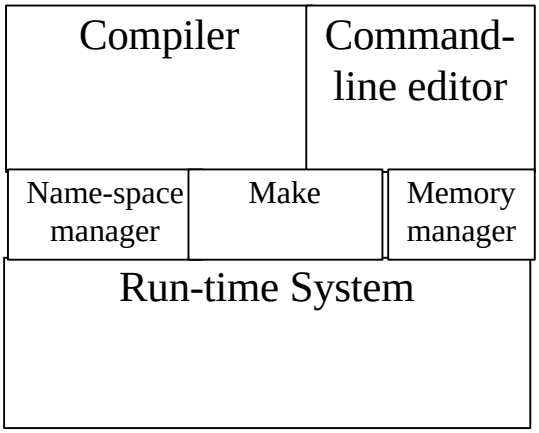
16.1 Platform

Barbados has been implemented in the Windows 95 platform. This means that it should port to the Windows NT platform with no modification to the code. It was written entirely in C, originally in the Borland C++ compiler and then in Microsoft Visual C++ v4.0.

16.2 Architecture

At present, the persistent system and programming environment are all contained together in the one executable program (with the exception of the 'LGO id server' described below) outside the Barbados persistent store. This organisation is not ultimately a satisfactory state of affairs, because the programming environment should be contained within the persistent store, rather than mixed in with the code that implements the persistent store. However, this was the easiest way to implement the functionality.

The main components of the programming environment are the following:



The command-line editor controls the system interface, that is the command-line editor, the scroll-buffer and the superficial lexical analysis used to identify compileable entities. The compiler compiles arbitrary strings. The name-space manager manages the name-space, i.e. does the actual work involved when a declaration is compiled, manages the 'current directory' and associated name-space, and deletes named objects too. 'Make' implements program consistency. The memory manager provides heap functions by building on top of Windows memory primitives. The run-time system includes the virtual machine-code interpreter and code for initialising system objects and providing system functions.

Control is most often in the command-line editor. Each time the user enters a compileable string, the string passes through the following components:

1. It is sent to the compiler.
2. The result is sent to the 'Make' system to be brought up-to-date prior to execution.
3. The run-time system is invoked, to interpret the generated virtual machine-code.

16.3 The Memory Manager

The memory manager must provide heap functions, i.e. malloc(), free(), realloc() and so on.

These functions had to be implemented anew because Barbados requires more control over memory than would be afforded with the standard library functions.

Specifically, the Barbados persistence model requires that each large-grained object which is currently open (i.e. in memory) belong in a heap of its own. It must be possible free an entire LGO in one go, efficiently, and to read in an LGO from disk and set it up in memory, efficiently. Also, Barbados needs to be able to map arbitrary pointers to the LGO which it corresponds to, and to annotate heap objects with extra information during the 'Saver' phase. Also, by implementing heaps myself, it was possible to perform extra checks e.g. that valid pointers are passed into the 'free()' and 'realloc()' functions.

16.4 The Compiler

The compiler was implemented in a quite standard way. Source-text passes through the following levels:

1. Lexical analysis (including preprocessor transformations)
2. Parsing (recursive descent), producing syntax trees.
3. Decoration of the syntax trees, to add type information and instruction information from the bottom up.
4. Optimisations of expressions (an optional piece of work which was implemented as an exercise).
5. Emission of instructions by traversing the syntax trees.

After compilation, the generated code is returned and declarations will have been processed. All other information, e.g. the program syntax trees, are thrown away. This differs to the approach taken by many incremental compilation researchers, who keep the syntax trees in order to make incremental modifications to them.

16.5 The Editor

The editor works in a standard way. It maintains a doubly-linked list of text lines, with a pointer to the current line. Each time the screen needs to be refreshed following a modification, a view of this data-structure is taken. An intermediate layer exists between the command-line editor and the Windows text-drawing functions; this layer optimises the changes to the window by identifying which lines have changed.

When a bitmap is output to the editor, it is (a) stored in a database of bitmaps the editor keeps, and (b) converted to a text string incorporating a special control character and an index into the database. This implementation enables the window-drawing code to display bitmaps.

16.6 The Name-Space Manager

The most important function implemented by the name-space manager is the 'DeclareObject()' function. This function takes (name, type, storage class) as parameters and declares the appropriate object.

This function is used to declare types, functions, variables and even preprocessor macros. It is used for global variables/objects, local variables/objects, class members, classes and enumerator-type elements.

It also implements functions to change the current directory. There are functions to add and subtract directories of objects to the current name-space.

16.7 Types used internally

The following types are used to represent program and data components in the system:

named_obj: A named object. Contains fields for type, value, storage class, make information, visibility (in the case of class members), and version (in the case of overloaded objects).

make_node: A node used by the 'make' algorithm. Contains fields for dependency lists and time-stamps. It always corresponds to one and only one named_obj.

directory_type: A linked list of named_obj's.

hash_type: A node corresponding to a particular (string) identifier. Can refer to multiple named_obj's, e.g. through overloading or through being declared multiple times in different (nested) scopes, or through existing multiple times in directories which are mapped into the name space.

type_type: A representation of a type. For example, "a pointer to array of 30 instances of class X'.

structure_type: An object corresponding to a struct or class definition. Contains a linked list of fields and an overall 'sizeof' figure.

16.8 Storing Dependencies

At present, dependency information is not stored persistently. This is because the information can be regenerated reliably simply by recompiling the relevant objects, and it was my hypothesis that better performance would result if this information was not stored but rather regenerated when needed.

The theory is that if a user wishes to use a stable program or library, they do not want to be slowed down by loading dependency information for the code. On the other hand, if they are developing a particular program or set of functions, the dependency information will remain in the Barbados cache for the majority of the development session.

Either alternative (to store dependencies in the store or not) is probably about as good as the other. In particular, once Barbados has a 'Release()' tool to bundle stable code into a single object and separate it from source-code, this will mean that dependencies need only be stored persistently for unstable code.

16.9 Detecting Compileable Entities:

A compileable entity, as far as the editor is concerned, is either:

- a piece of text which includes a '{' token that is matched further down, or:
- a piece of text without any braces but which has a semi-colon token, and the semi-colon does not occur inside '(' and ')' brackets[9], or
- a piece of text beginning with the '#' symbol and finishing with a line that does not end in the '\' character.

The editor knows about the lexical structure of C++, and hence is not fooled by semi-colons and braces inside comments or strings or preprocessor directives.

For example:

[9] semi-colons occur inside the '(' and ')' brackets in 'for' loops.

void f(int n) { while (n--) { cout << n; } }	*Braces match. The final '}' completes it.*
cout << "Hello.";	*No braces but a semi-colon.*
while (n--) cout << n;	*No braces but a semi-colon.*
4+3; 4+3+2;	*Actually, two compileable entities.*

By comparison, these examples are not complete:

for (int i=0; i < 5; i++)	*The semi-colons are inside brackets, so don't count.*
cout << "Hello;"	*There is no semi-colon token. (The semi-colon in the string doesn't count).*
int f(int n) /* { } */	*The braces are in a comment.*

While this algorithm is highly effective, it can be fooled:

`#define close_brace }` `void f(int n)` `{` `  return 5;` `close_brace`	*This unusual macro confuses the half-parser and it doesn't recognise that the entity is complete.* *Some non-syntactic uses of macros can cause trouble.*
`4+3;  cout` `  << "Hello.";`	*The semi-colon means that something is complete, but the compiler will be called before the user can enter the next line.*
`if (true)` `  cout << "yes";` `else cout << "no";` `while (--n)` `  if (n % 2 == 0)` `  cout << n;` `  else cout << n+1;`	*The compiler is called after the "yes", because up to there it is syntactically correct, and it never even sees the 'else'. The first example is an unusual thing to do in an interpreter, (the user can just evaluate the condition themselves and save themselves some typing); the second example is a more valid example. In this case, the user has to put braces in somewhere.*

`int A[5] = {` `    2,3,2,3,2 }` `;` `class C {` `    int x;` `    ...` `}` `a,b,c;`	*Declarations that use braces, i.e. array initialisation and class definitions, must have the closing semicolon on the same line as the closing brace, or the half-parser will be confused.*

However, it is easy to find ways of coping with these problems. In most cases, the user can work around the problem simply by putting braces around a command.

The editor maintains all relevant information incrementally, as the user moves up and down over text.

17. Appendix C: The Meta-Level Classes' Interfaces

The meta-level classes' interfaces as described in chapter 6 are given below:

```
class any {                                 // A (type, value) pair
      ...
public:
      type       Type();                    // type
      void*      Address();                 // value
      any        ArrayLookup(int);          // Accessing
      int        ArrayLength();
      any        FieldLookup(string s);
      any        FieldLookup(named_obj obj);
      any        FunctionCall(any *Params);
      any        Dereference();
      ostream&   Print(ostream&);           // Display
};


struct type_expansion {                     // Used for expanding types.
      char       primary;                   // ptr,array,fundamental etc
      type       remainder;                 // ptr etc. to what?
      int        n;                         // Array size or fn arity
      type       params;                    // Types of parameters
      named_obj  obj, typedef_obj;          // Root of list of members
};


class type {                                // A run-time type
      ...
public:
      char       Expand(type_expansion*); // Expand to 'type_expansion'
      type       NextParam();             // Cycle thru fn params
      type       operator=(string s);     // Assign
      any        New();                   // Create a new instance
      ostream&   Print(ostream&);         // Display it
};


class named_obj {                           // A directory entry or class
member
      ...
public:
      named_obj(string s);                // Find in current name-space
      named_obj  Next();                  // Iterate thru directory or
class
      string     Name();                  // The name
      any        Any();                   // The (type, value)
      type       Type();                  // The type
      boolean    Exists();                // Test for being null
      named_obj  Dependency(int n);       // Lists dependencies
      ostream&   Print(ostream&);         // Display
};


class directory {                           // A set of named_obj's
      ...
public:
      any        FindAny(string);         // Find this as an any
```

```
        named_obj  Find(string);            // Find this as named_obj
        named_obj  Create(string, type);    // Create a new named_obj
        void       Delete(string);          // Delete this named_obj
        ostream&   Print(ostream&);         // Directory listing
};
```

18. References

[Aho86]: Aho A.V., Sethi R. & Ullman J.D., “Compilers, Principles, Techniques and Tools”, *Addison-Wesley*, Reading, Massachusetts, 1986.

[Atk83]: Atkinson M.P., Bailey P.J., Chisholm K.J., Cockshott P.W., Morrison R. "An Approach to Persistent Programming", *The Computer Journal, Vol 26,* 1983, pp 360-365

[Atk86]: Atkinson, M.P., Morrison R., “Integrated Persistent Programming Systems”, *Proceedings of the 19th International Conference on System Sciences*, Hawaii, pp 842-854, 1986.

[Atk95]: Atkinson M.P. & Morrison R., “Orthogonally Persistent Object Systems”, *VLDB Journal 4*, 3 1995, pp 319-401.

[Bah86]: Bahlke R. & Snelting G., “The PSG System: From formal language definitions to interactive programming environments”, *ACM TOPLAS,* 8 (4): pp 547-76, 1986.

[Bor88]: Borras P. et al, “Centaur: the System”, *Proceedings of SIGSOFT’88, 3rd Annual Symposium on Software Development Environments (SDE3)*, Boston, USA, 1988.

[Bro92]: Brown A.L. & Morrison R., “A Generic Persistent Object Store”, *Software Engineering Journal 7*, 2 (1992) pp 161-168.

[Buh86]: Buhr P., “A Programming System”, *PhD Thesis*, University of Waterloo, 1986.

[Buh89]: Buhr P., Ditchfeild G. & Zarnke C.R., “Basic Abstractions for a Database Programming Language”, *2nd International Workshop on Database Programming Languages*, June 1989, pp 422-437.

[Car94]: Carey, M.J. et al, “Shoring Up Persistent Applications”, *Proc. 1994 ACM-SIGMOD Conference on the Management of Data*, 1994.

[Coo95]: Cooper T. & Wise M., “The Case for Segments”, *Proceedings of IWOOOS ‘95*

[Coo96a]: Cooper, T. & Wise, M., “Barbados: An Integrated Persistent Programming System”, *unpublished,* http://www.cs.su.oz.au/~timc/Barbados, 1995.

[Coo96c]: Cooper, T. & Wise, M., “Critique of Orthogonal Persistence”, *to be published in Proceedings of IWOOOS ‘96, also available from:* http://www.cs.su.oz.au/~timc/Barbados, 1996.

[Dah87]: Dahle H.P., Lofgren M., Madsen O.L. & Magnusson B., “The Mjolner Project”, *Proceedings of the Conference held at Software Tools*, Online Publications, London, 1987.

[Dea94]: Dearle A., di Bona R., Farrow J., Henskens F., Lindström A., Rosenberg J. & Vaughan F., "Grasshopper: An Orthogonally Persistent Operating System", *Computer Systems*, Vol 7, 3, pp. 289-312, 1994.

[Don80]: Donzeau-Gouge V., Huet G., Kahn G. & Lang B., “Programming Environments based on Structured Editors: the Mentor experience”, *Rapports de Recherche* 26, INRIA, Domaine de Voluceau, Rocquencourt, France, 1980

[Dub93]: DuBois, Paul, “Software Portability with imake”, O’Reilly and Associates, 1993.

[Ear72]: Earley J. & Caizergues P., “A method for incrementally compiling languages with nested statement structure”, *Communications of the ACM,* 15 (12) pp 1040-1044. Dec 1972.

[Fel79]: Feldman S.I., “Make - A Program for maintaining Computer Programs”, *Software - Practice and Experience* 9(3) pp255-265, 1979.

[Far92]: Farkas A.M., Dearle A., Kirby G.N.C., Cutts Q.I., Morrison R. & Connor R.C.H., “Persistent Program Construction through Browsing and User Gesture with some Typing”, *Proc. 5th International Workshop on Persistent Object* Systems, San Miniato Italy (1992) pp 375-394.

[Fra83]: Fraser C. & Hanson D., "A High-Level Programming and Command Language", *ACM Transactions on Programming Languages and Systems,* Vol 5, 1983

[Hee85]: Heering J & Klint P., "Towards Monolingual Programming Environments", *ACM Transactions on Programming Languages and Systems,* Vol 7, No 2, April 1985

[Hee94]: Heering J., Klint P. and Rekers J., “Lazy and Incremental Program Generation”, *ACM TOPLAS* 16(3), pp 1010-1023, 1994.

[Hen93]: Henskens F.A., Brossler P., Keedy J.L. & Rosenberg J., “Coarse and Fine Grain Objects in a Distributed Persistent Store”, *Proceedings of IWOOOS ‘93*, pp 116-123

[Hos90a]: Hosking A.L., Moss J.E.B., "Towards Compile-time Optimisations for Persistence", *Implementing Persistent Object Bases (4th Workshop on Persistent Object Systems),* 1990

[Hos90b]: Hosking A.L., Moss J.E.B. & Bliss C., "Design of an Object-Faulting Persistent Smalltalk”, Computer Science Technical Report 90-45, University of Massachusetts, Amherst, 1990

[Gus89]: Gustavsson A., “Viewing the Evolution of Software Objects in an Integrated Environment”, *Proc. 2nd ACM SIGSOFT Workshop on Configuration Management,* Nov 1989.

[Joy92]: Ian Joyner, "C++?? A Critique of C++, 2nd Edition", *Unpublished, available upon request to ian@syacus.acus.oz.au*

[Ker84]: Kernighan B.W. & Pike R., “The UNIX Programming Environment” *Prentice Hall Inc.*, 1984.

[Kir92]: Kirby G.N.C., Connor R.C.H., Cutts Q.I., Dearle A., Farkas A.M. & Morrison R., “Persistent Hyper-Programs”, *Proc. 5th International Workshop on Persistent Object* Systems, San Miniato Italy (1992) pp 73-95.

[Kir92b]: Kirby G.N.C., “Persistent Programming with Strongly Typed Linguistic Reflection”, *Proc. 25th International Conference on System Sciences, Hawaii* (1992) pp 820-831.

[Kir93]: Kirby Graham N. C., "Reflection and Hyper-Programming in Persistent Programming Systems", (*PhD Thesis - University of St Andrews),* 1993

[Lam91]: Lamb C.W., Landis G., Orenstein J.A. & Weinreb D.L., “Next Generation Database Systems”, *CACM* 43,10 (Oct 1991),pp 51-63.

[Lal90]: LaLonde W.R. & Pugh J.R., “Inside Smalltalk”, *Prentice-Hall,* 1990.

[Lie93]: Liedtke J., “A Persistent System in Real Use - Experiences of the First 13 Years”, *IWOOOS ‘93*, pp 2-11, 1993.

[Lin95]: Lindstrom A., Rosenberg J. & Dearle A., “The Grand Unified Theory of Address Spaces”, *Proceedings of 5th workshop on Hot Topics in Operating Systems,* 1995.

[Mag90]: Magnusson B., Bengtsson M., Dahlin L., Fries G., Gustavsson A., Hedin G., Minor S., Oscarsson D. & Taube M., “An Overview of the Mjolner Orm Development Environment: Incremental

Language and Software Development", *Proceedings of 2nd International Conference TOOLS (Technology of Object-Oriented Languages and Systems)*, 1990.

[Mar90]: Marlin C., "A Distributed Implementation of a Multiple View Integrated Software Development Environment", *Proceedings of 5th Conference on Knowledge-Based Software Assistants*, pp 388-402, New York, 1990.

[Mcc96]: McCarthy M. J., "Incremental Code Generation in a Distributed Integrated Programming Environment", *PhD Thesis*, Flinders University, South Australia, 1996

[Mey88]: Meyer Bertrand, "Object-oriented Software Construction", Prentice Hall, 1988.

[Med81]: Medina-Mora R. & Feiler P.H., "An incremental programming environment", *IEEE Transactions on Software Engineering*, 7(5), pp472-482, Sep 1981.

[Mic96]: Henshaw J., "Inside Visual C++'s Incremental Build Technologies", http://www.microsoft.com/VISUALC/V4/V4tech/incrbld.htm

[Mor88]: Morrison R., Brown A.L., Conner R.C.H. & Dearle A., "Napier88 Reference Manual", *Universities of Glasgow and St Andrews*, Persistent Programming Research Report PPRR-77-89, 1989

[Mor89]: Morrison R., Brown A.L., Conner R.C.H. & Dearle A., "Napier88 Reference Manual", *Universities of Glasgow and St Andrews*, Persistent Programming Research Report PPRR-77-89, 1989

[Mor93]: Morrison R., Baker C., Connor R.C.H., Cutts Q.I. & Kirby, G.N.C. "Approaching Integration in Software Environments", *University of St Andrews Technical Report CS/93/10* (1993)

[Mor94]: Morrison R., Baker C., Connor R.C.H., Cutts Q.I., Kirby Q.N.C. & Munro D., "Delivering the Benefits of Persistence to System Construction and Execution", *Proc. 17th Australasian Computer Science Conference, Christchurch NZ* (1994) pp 711-719

[Mor95]: Morrison R., Connor R.C.H., Cutts Q.I., Dunstan V.S. & Kirby G.N.C., "Exploiting Persistent Linkage in Software Engineering Environments", a*ccepted by Computer Journal*, 1995.

[Nap88]: Morrison R., Brown A.L., Conner R.C.H. & Dearle A., "Napier88 Reference Manual", *Universities of Glasgow and St Andrews*, Persistent Programming Research Report PPRR-77-89, 1989

[Not85]: Notkin D., "The Gandalf Project", *Journal of Systems and Software*, (5) pp 91-106, May 1985.

[Ora91]: Oram A. & Talbot S., "Managing Projects with Make", O'Reilly and Associates, 1991.

[Pla92]: Plainfosse D. & Shapiro M., "A Distributed GC in an Object-support Operating System", *The 1992 Int. Workshop on Object-Orientation and Operating Systems (France)*, pp221-229, Oct 1992.

[Rep85]: Reps T. & Teitelbaum T., "The Synthesizer Generator", *ACM SIGPLAN Notices* 19 (5), pp 42-48, May 1984.

[Ric89]: Richardson, J.E. and Carey M.J., "Implementing Persistence in E", *Proceedings of the Third International Workshop on Persistent Object Systems*, Newcastle, Springer-Verlag, pp 175-199, 1989.

[Ric93]: Richardson J.E., Carey M.J. & Schuh D.T., "The design of the E programming language", *ACM Transactions on Programming Languages and Systems*, 15(3) July 1993.

[Ros85]: Rosenberg J. & Abramson D.A., "MONADS-PC: A Capability Based Workstation to support Software Engineering", *Proceedings of 18th Hawaii International Conference on System Sciences*, pp. 515-522, 1985.

[Sha89a]: Shapiro Marc, "Persistence and Migration for C++ Objects", *The European Conference on Object Oriented Programming (ECOOP)*, 1989, pp 191-204.

[Sha89b]: Shapiro M. et al, "SOS: An Object-Oriented Operating System - Assessment and Perspectives", *Computing Systems 2(4)* pp 287-337, 1989.

[Sha92]: Shapiro M., Dickman P. & Plainfosse D., "Robust, distributed references and acyclic garbage collection", *Symposium on Principles of Distributed Computing (Canada)*, 1992.

[Sin92]: Singhal V., Kakkad S. & Wilson P., "TEXAS: An Efficient, Portable, Persistent Store", *Proceedings of the 5th International Workshop on Persistent Object Systems*, 1992, pp 13-28

[Sou94]: Sousa P. & Marques J.A., "Object Clustering in Persistent and Distributed Systems", *Proceedings of the Workshop on Persistent Object Systems 1994, France.*

[Tei81]: Teitelbaum T. & Reps T., "The Cornell Program Synthesizer: A syntax-directed Programming Environment", *Communications of the ACM*, 24 (9) Sep 1981, pp 563-573.

[Tic85]: Walter F. Tichy, "RCS - A System for Version Control", *Software Practice and Experience 15*, 7, (July 1985), pp 637 - 654.

[Tic86]: Tichy, W.F., "Smart Recompilation", *ACM TOPLAS* 8(3), 1986.

[Weg89]: Wegner P., "Dimensions of object-based language design", *OOPSLA 87*, 1987.

[Wil84]: Wilander J., "An Interactive Programming System for Pascal", *Interactive Programming Environments*, McGraw-Hill, pp 117-127, 1984.

[Wir92]: Reiser M. & Wirth N., "Programming in Oberon: Steps Beyond Pascal and Modula", *ACM Press Books*, 1992.