Amadeus Project

C** Programmers' Guide

Distributed Systems Group

Trinity College Dublin

Abstract

The C** extensions to C++ for distributed and persistent programming in C++ are described.

Document Identifier	Release Doc:3
Document Status	Orange2.1
Created	4/Feb./1992
Revised	
Distribution	Public

Contents

Preface

The Amadeus v2.1 (*orange*) release is an upgrade of the second release from the Amadeus project by the Distributed Systems Group of the Department of Computer Science, Trinity College Dublin (TCD). The orange release follows on from the earlier red release in July 1991. Amadeus extends C++ for distribution and persistence; the extended language is called C**.

Amadeus v2.1 is itself implemented in C and C++, on top of Digital Equipment Corporation's Ultrix 4.3 on both MicroVAXes and DECstations, and on SunOS 4.0 on Sun-3s, and Sun-4s.

The C** compiler in Amadeus v2.1 is a modified version of the Free Software Foundation's g++ 1.37 compiler.

Future releases, both planned and underway, will extend the functionality, supported languages and host operating systems.

The Amadeus Project has been influenced by the Esprit Comandos-1 and Comandos-2 projects, in which TCD has been a partner. The project is also being influenced by the Esprit Harness and Ithaca projects. TCD acknowledges the fruitful interactions with all of the participating institutions and in particular with Inesc in Lisbon; Bull and IMAG in Grenoble; GMD in Bonn; and the University of Glasgow.

Information Set

The information set for the Amadeus v2.1 release includes the following three documents:

- *Overview of the Amadeus Project* presents an overview of both the Comandos and Amadeus projects. This document also includes an introduction to distributed systems and explains the terminology used in the Amadeus Project.

- *C** Programmers' Guide* contains both tutorial and reference information on programming in the C** language. The text assumes a knowledge of object-oriented programming and, in particular, the C++ language. This guide lists sample C** programs which are supplied with Amadeus v2.1, and which can be compiled and linked after installing the Amadeus environment.

- *Amadeus Installation and User Guide* contains the installation procedure together with instructions for startup, and configuration.

Organization of This Document

The intended audience for *C** Programmers' Guide* is applications programmers who are knowledgeable in C++, but possibly having only a limited knowledge of distributed processing. The document is arranged in four chapters:

Chapter 1 provides a backgound to C** and some of the guidelines which influenced its design. The chapter may perhaps be skipped on a first reading of the document.

Chapter 2 illustrates the use of C** via some simple (and trivial) examples.

Chapter 3 describes the details of the extensions made to C++ by C** .

Chapter 4 describes the interface to functions available directly from the Amadeus run-time.

Related Texts The following texts are recommended as background reading on the C++ Language:

- *The Annotated C++ Reference Manual*, Bjarne Stroustrup, (1990), Addison Wesley
- *Users's Guide to GNU C++ (version 1.37)*, Michael D. Tiemann, GNU C++ release

Trademarks The following are trademarks:

- DECstation – Digital Equipment Corporation
- microVAX-II – Digital Equipment Corporation
- Network File System – Sun Microsystems, Inc
- NFS – Sun Microsystems, Inc

- Ultrix – Digital Equipment Corporation
- UNIX - UNIX Systems Laboratories, Inc
- VAX – Digital Equipment Corporation

Chapter 1

Background to C**

The overall objective of the Amadeus Project is to design and develop an integrated application support environment for writing and running distributed applications which can manipulate persistent (long-lived) data. This release of Amadeus (v2.1) is a step towards meeting this overall goal by providing support for C++ in a distributed and persistent environment. Support will be extended to other Unix variants and well known programming languages in future releases.

A goal of the Amadeus project is to extend each language as little as possible. This has two major benefits: programmers do not require extensive retraining to use the Amadeus environment and existing code can be re-used as much as possible.

Ideally, no language extensions would be necessary to write applications that run in the Amadeus environment. In practice, however, interfacing a language to the Amadeus environment requires that additional information be provided at runtime. Typically, a preprocessor is provided for each supported language which generates the necessary supplementary information and then calls the native compiler. A further practical concern is that it is not always possible to support all the features of a given language in Amadeus' distributed and persistent environment.

Amadeus v2.1 supports C++ with extensions; this extended language is called C**. Rather than using a preprocessor, the changes to support C** have been integrated into a C++ compiler.

An introduction to the Amadeus environment is given in the *Overview of the Amadeus Project*. In summary, the environment is intended to provide the necessary low-level support for distributed and persistent programming, including:

- Transparent access to local and remote objects;
- Transparent object storage and retrieval;
- Object location services;
- Garbage collection;
- Dynamic link loading;

- Object sharing;

In Amadeus 2.1, garbage collection is limited, and dynamic link loading has not been supported.

The services above are also intended to be supported in a range of programming languages. However the only language supported by Amadeus v2.1 is C**.

The Amadeus environment itself consists of two components: the *generic runtime*, and the *Amadeus kernel*. The generic runtime is layered above the Amadeus kernel, and provides a language-independent method of accessing remote objects. The normal UNIX services are, in principle, still available in the Amadeus environment. Likewise C** is an upwards compatible derivative of C++: any C++ program will continue to execute in the Amadeus environment for C** . The Amadeus environment is intended to be language independent, so that both existing and new languages can be used to write distributed and/or persistent applications.

1.1 Considerations which influenced C**

In extending a language to execute above the Amadeus environment, we believed it important to reflect the philosophy of each particular language, as far as is reasonable.

In the case of C++, we felt that an important theme of the language is that C++ programmers "only pay for those features which they use." Thus we rejected the option to provide persistence and distribution support *automatically* for *every* C++ object: rather we felt it important that this support should be selectable by the programmer.

A further theme is "the compiler assumes the programmer knows what she's doing." Thus we felt it important not to disallow, for example, type casts which may compromise the support we provide.

We also felt it important that, as far as possible, all existing code should continue to work. However in re-using existing code in the Amadeus environment, clearly one must be aware of the consequences. For example, Amadeus provides a system wide (distributed) persistent store. However `static` storage, as in C++, is of course not persistent: updating a `static` value, and then re-running your program will cause the `static` value to revert to its initial value. Further, the same executable file may be run simultaneously in different Amadeus clients, but the `static` storage of these is *not* shared. These points may cause the novice C** programmer some confusion, since she may expect to be able to communicate values through `static` storage, persistently to other programs. In summary, `static` storage "works" the same way in C** as in C++: one must do additional programming to exploit persistence and distribution.

We felt it important to use inheritance as much as possible. Like several other projects, and certain public domain libraries, in C** persistence can be added to a class and inherited by its derived classes. In this way, persistence can readily be added to a collection of classes. It should be carefully noted that a persistent class can have *both* persistent and non-persistent instances: whether a given object from a persistent class is actually persistent, depends (dynamically) on whether other persistent objects refer to it. Correspondingly, a non-persistent class – eg an "ordinary" C++ class – can *never* have any persistent instances. Persistent classes incur overheads in additional code and space, as will become clear in section **??**.

In respect of supporting remote use, we found it difficult to exploit inheritance as we do for persistence. Remote use is an attribute of individual members[1] of a class interface, and not directly of the instances of that class. We also believe it natural to treat remote use separately from persistence: whether or not a class supports remote use of some or all of its members, it may or may not support persistent instances. Naturally, a member which can be used remotely can also be used locally. Further, members which support remote use can be inherited by derived classes. As a result when using such a member, it is normally transparent to the programmer whether the designated object is actually local or remote; further, the proximity of the object may change dynamically at runtime. Classes which support remote use of their (possibly inherited) members incur overheads in additional code and space, as will become clear in section **??**.

Clearly adding support for distribution to any language increases the range of possible exception conditions which can arise. While waiting for a stable implementation of exceptions in C++[2], we felt it wisest to have a simple implementation of recovering from exceptions in the Amadeus environment. We therefore only allow programmers to specify a single function to be called whenever an environment exception is raised.

For concurrency, we have extended C++ with a simple "futures" mechanism which allows asynchronous invocations to be started in parallel with the creating thread of control. The creating thread of control may choose to synchronise with the result of the asynchronous invocation at a later stage: or it may choose to ignore that result.

For synchronisation, we have not made any particular effort to extend C++. Instead, the Amadeus environment primitives for these facilities are made available to the C** programmer.

We believe it ought to be possible to support various C++ libraries which add concurrency and synchronisation support to C++ – but so far we have not attempted to do this.

[1]Here we use the term "members" in the C++ sense – i.e. the operations and instance data declared by a class. Eiffel programmers would use the term "features".

[2]Currently we use gnu 1.37, whose exception model does not correspond to that suggested in the Annotated C++ Reference Manual.

Chapter 2

Short introduction to C**

C** is an extension of the C++ programming language suitable for writing programs which run in a persistent distributed environment.

We start this chapter with a trivial example program which illustrates how a C++ class can be extended for persistence and remote use in C**. We then proceed to show how to compile and execute the program. Finally we give a more elaborate example, illustrating more of the features of C**.

2.1 Trivia

2.1.1 Persistence

Consider the following C++ program:

```
#include <stream.h>

class integer {
           int value ;
public:
           integer (int Value=0) { value = Value ; };
           int operator ()() { return value ; };
           void inc () { value++ ; };
} ;

main()
{ integer *i;

  i = new integer ();
  cout << "i is " << (*i)() << "\n";
  i->inc();
}
```

Clearly, if we compiled this program as "foo" and ran it twice, we would obtain:

```
% foo
i is 0
% foo
i is 0
%
```

We can compile and run this same program using C** too, and (naturally) obtain the same result. However, we can alter the behaviour of class `integer` so that its instances can be *persistent*. In programming environments, it is often necessary to manipulate many persistent objects, such as application data, source code, interface specifications, and compiled code. The advantage of providing persistence for programmers working in an object-oriented environment is that pointers to objects do not have to be rebuilt every time the objects are retrieved from storage.

If we make class `integer` persistent (in the manner explained later below), then when we run the (new) foo, we will obtain[1]:

```
% foo -reset
i is 0
% foo
i is 1
% foo
i is 2
%
```

Note that there is an optional command line switch `-reset`. If given, it indicates that the application (here `foo`) should do a full "initialisation", based on an empty persistent store. If not (the default), it indicates that the application should resume from the stored state which resulted from the previous execution. The `-reset` switch, if given, must be the first command line option; further it is *not* passed to the `main` of the application, but instead filtered out by the Amadeus runtime. `-reset` can be given again at any time:

```
% foo -reset
i is 0
% foo
i is 1
% foo
i is 2
% foo -reset
i is 0
%
```

In order to make our class `integer` persistent, we can either derive a persistent class from it, or re-write it. Both options are explored below.

[1]Note that the Amadeus server daemon must be running on your machine before executing any applications. The *Amadeus Installation and User Guide* explains how to run and interact with this daemon, including control of system level debugging messages if required.

2.1.1.1 Persistence via a permclass

In C**, a persistent class is any `permclass` or any class which inherits (directly or indirectly) from any other `permclass`[2]. `permclass` is a new keyword introduced in C**.

Thus, to define persistent integers, we just change `class` to `permclass` and include the Amadeus library[3]:

```
#include <amadeus.h>
#include <stream.h>

permclass perm_integer {
           int value ;
public:
           perm_integer (int Value=0) { value = Value ; };
           int operator ()() { return value ; };
           void inc () { value++ ; };
} ;

main ()
{ perm_integer *i;

  i = new perm_integer ();
  cout << "i is " << (*i)() << "\n";
  i->inc();
}
```

Note that, as in this example, an object whose value is to persist should always be allocated on the heap using `new`. Any other allocation is permissible, but the lifetime of the object is the same as it would be for the normal C++ semantics. For example, an `automatic` object will have its lifetime limited by the enclosing block scope; and updates to the value of a `static` object will not persist across program runs.

2.1.1.2 Persistence by derivation

An alternative mechanism would be to introduce a `permclass` as a derived class of `integer`[4]:

```
#include <amadeus.h>
#include <stream.h>

class integer {
           int value ;
public:
           integer (int Value=0) { value = Value ; };
           int operator ()() { return value ; };
```

[2] However see also section ??.

[3] In practice, persistent objects are usually also registered with the name service, c.f. section ??.

[4] In practice, persistent objects are usually also registered with the name service, c.f. section ??.

```
          void inc () { value++ ; };
} ;

permclass perm_integer_v2 : public integer{};

main ()
{ perm_integer_v2 *i;

  i = new perm_integer_v2 ();
  cout << " i is " << (*i)() << "\n";
  i->inc();
}
```

2.1.1.3 `main`

As in C++, every C** program should contain a function `main` which is called as the entry point of the program (in fact it is called after any static constructors have been executed). As noted in section **??**, if the `-reset` switch is given in the command line, it is filtered out from the standard `argv` and `argc` values before `main` is called.

The Amadeus runtime supplies a call[5] `int amadeus.reset()` which can be used (e.g. in `main`) to determine whether the `-reset` switch was given in the command line:

```
#include <amadeus.h>
....
  if (amadeus.reset())
        // -reset was given
  else
        // -reset was not given
```

2.1.1.4 Binding to the Persistent Store

In Amadeus every object whose class is a `permclass` (base or derived) is said to be potentially persistent. Specifically, it persists if it itself is registered in an Amadeus Name Store, or if it is (transitively) referenced by a pointer from an object so registered. If the object is not registered or referenced in this way, it will (eventually) be discarded by the Amadeus garbage collector. Note that for an object to persist it does not necessarily need to be registered itself in a Name Store – it suffices for it to be referenced by any other persistent object.

In Amadeus v2.1 above Unix, an Amadeus Name Store is implemented by directly using the Unix file system. Each object registered in the Name Store is given a Unix file name by calling the function:

```
  void amadeus.record(const char *Unixfilename,  void *object);
```

Likewise the Name Store can be consulted so as to recover a registered object:

[5] A "down" call in Amadeus terms.

```
  void* amadeus.lookup(const char *Unixfilename);
```

The result of a lookup is a `void*`, which usually must be (perhaps implicitly) typecast to a pointer to an appropriate class. In Amadeus v2.1, there is no direct support for asserting that such a type cast is safe.

Note that the programmer does not need to explicitly save or fetch a persistent object; when called, `record` and `lookup` do *not* themselves cause an object to be moved to or from the persistent store. Instead they merely register a binding from an ASCII text string (i.e. a Unix path specification) to a system wide object identifier[6].

Using the Name Store, the example of a persistent integer (c.f. section **??**) can be completed:

```
main()
{ perm_integer *i;

  if (amadeus.reset())
    { i = new perm_integer();
         amadeus.record ("myinteger.ns", i); }

  // normal case
  if (! (i = amadeus.lookup ("myinteger.ns")))
   { cout << "Object lost from name store!\n"; exit(-1); }

  cout << " i is " << (*i)() << "\n";
  i->inc();
}
```

2.1.2 Remote use

As well as extending C++ with support for persistence, C** allows specified member functions of a class to be used remotely. The keyword `global` is used to mark such member functions. Member objects which themselves have global functions can also be used remotely (c.f. section **??**).

Returning to our integer example, we could mark member functions for remote use[7]:

```
class system_wide_int {
           int value ;
public:
           system_wide_int (int Value=0) { value = Value ; };
           global int operator ()() { return value; };
           global void inc () { value++ ; };
} ;
```

To create an integer of this class for remote use:

[6] And currently we rely on NFS to maintain such bindings. Clearly other name services, such as DCE CDS could in principle be used as an alternative.

[7] In Amadeus v2.1, C** constructors cannot be used remotely.

```
system_wide_int *i = new system_wide_int();
```

The pointer to the new object can subsequently be communicated to other nodes in the distributed system. The pointer can then be used to access the object from any point in the distributed environment.

```
cout << "i is " << (*i)() << "\n";
```

Thus in some sense, the new object (the `system_wide_int`) we created above can act as a "server" for any "client" in the distributed network who has the value of the pointer.

Thus the use of a pointer to a remote object in C** is syntatically *identical* to the usual use of a pointer in C++: the current location of a (potentially remote) object is not apparent from a pointer to it. Indeed, the location of the object may change at runtime, transparently to other objects holding pointers to it.

Coupled with the ability to support persistent objects, Amadeus and C** provide a powerful and yet elegant environment in which to build distributed C++ programs.

2.2 Compiling and linking a C** application

The C** compiler is a modified version of the gnu `g++` v1.37 compiler. The compiler itself is called, as usual, after `cpp`[8] has processed the input source. The `cpp` has not been modified for C** .

Every `permclass`, or class with `global` members, requires the C** compiler to generate additional code. This code is generated as C++ source into a single file with the prefix "." and extension "`.pr`", when the `-proxy` command line switch is enabled. Note that the "." prefix results in the generated files not normally being visible in the directory when a simple `ls` command is issued[9].

Thus, given a source file `foo.cc` (perhaps containing one of the `integer` class examples):

```
% css -proxy foo.cc
```

compiles `foo.cc` into `foo.o`, as usual, but *also* generates a new file `.foo.pr`.

Note that the `-proxy` switch can be used on source files which do not contain any `permclass`es or classes with `global` members, although doing so is superfluous. If however this is done, then additional `.pr` files are generated which in fact are not needed.

The source file which contains the definition of `main(...)` (i.e the main entry point of the program) must also be compiled with the `-proxy` switch. This is because `main` is currently "mangled" by the C** compiler to `aon_main` so as to overcome some problems in constructor initialisation in g++.

In addition to generating `.pr` files for the various source files of an application, three files common to the whole application are generated:

- `.class-dictionary` – a list of class names, their source files, and an `unsigned int` number for each class.
- `.regclasses.pr` – a C++ source file containing code to register classes with the Amadeus environment during initialisation.
- `preprocessed.cc` – a file which will include all of the `.pr` files generated for the application when compiled.

The `preprocessed.cc` must also be compiled, but not until *after* all the original C** source files have been compiled with the `-proxy` switch:

```
% css preprocessed.cc
```

Finally, the various `.o` files should be linked with the standard Amadeus library:

```
% css -o foo foo.o preprocessed.o -L/usr/lib/gnu/lib libpga.a
```

[8]or rather, gcc-cpp.

[9]i.e. use `ls -a` to see these "hidden" files.

Naturally, the sequence of compiles is usually captured in a `Makefile`. Further, the Amadeus library and associated header files are unlikely to be available in your current directory and thus, for example, the `-I` switch must be used. The sample programs included in the Amadeus C** distribution each include such a `Makefile`, which can be used as models for further ones.

2.2.1 Running a C** application

Running any Amadeus application requires that an Amadeus server daemon be running on your host(s), and that certain configuration files are also available. Details of these issues are given in the *Amadeus Installation and User Guide*.

2.2.2 More details on compilation options

This section can be omitted on a first reading of this Guide. It describes further flags which can be indicated for a compilation, and is included at this point in the Guide for completeness.

2.2.2.1 Concurrent access to the Class Dictionary

The class dictionary, as indicated above, is shared between the multiple compiles of an application. In some cases, it may be desirable to launch several compiles in parallel[10]. In this case, simultaneous compiles may attempt to access the class dictionary. In general the class dictionary is appended to during a compilation, as well as read. Thus attempted concurrent access must be serialised.

If the `-lock` switch is given as a command line option, the C** compiler will use the `lockf(3)` call to control concurrent access to the class dictionary. The compiler uses `lockf(3)` only when accessing the class dictionary, and releases each lock as soon as possible thereafter, so reducing contention to the shared file.

The compiler defaults to not using `lockf(3)`.

2.2.2.2 Sharing a common Class Dictionary across several applications

It is often essential to be able to share a set of header files across several applications. Doing so in C** usually also requires that a common class dictionary file be used.

The default file for the class dictionary (`.class-dictionary`) can be overruled by using the `-CD` switch. For example specifying `-CDa/b/c` indicates that the file `a/b/c` should be used as the class dictionary.

When using the `-CD` switch, new classes, introduced by the current application, are registered in the common class dictionary. However the file `.class-dictionary` is still used by C** so as to indicate which entries in the common class dictionary are needed by the current application.

[10]For example, by using a distributed `make` facility.

Specifying `-CD.class-dictionary` is illegal.

If a class dictionary is being shared among several programmers, it is probably advisable to also use the `-lock` switch.

2.3 A moderate example – The Whiteboard Demonstration

The whiteboard demonstration is a distributed, persistent application built on top of Amadeus. It consists of a number of objects called whiteboards which are stored in a directory object. The user may create new whiteboards, specifying a name, and enter them into the directory. A whiteboard may be viewed by a screen object. Once a whiteboard is registered in the directory, users may enter text into the whiteboard by specifying its name and a text string. Similarly, a whiteboard may be viewed or deleted.

The whiteboard example is a part of the orange release of Amadeus.

2.3.1 How to Run the Demonstration

The application is cold started by choosing a node number to provide a logical container for storage for the application – e.g. node 1[11], Then type:

```
wbserver -reset 1
```

which initiates the application using node 1 for storage. The application may now be warm started by typing

```
wbserver
```

and the prompt

```
Enter Command:
```

All input is prompted for, and a menu appears if unrecognised input is given. The menu is:

```
s - select and display a whiteboard
r - create a whiteboard
c - change to another whiteboard
d - delete a whiteboard from the directory
l - list the contents of the directory
t - enter text into a whiteboard
e - exit
```

[11]Note that the Amadeus server daemon must be running on your machine before executing any applications. The *Amadeus Installation and User Guide* explains how to run and interact with this daemon.

2.3.2 Implementation Details

When a user runs the application, a screen object is created locally on the current node. An I/O object is also created which runs a menu also on the local node. This provides the user with a choice of commands to create and manipulate whiteboards. Neither the screen nor the I/O objects are persistent: each time the application is run a new screen and I/O object are created.

On a cold start (-reset is specified as a command line argument), a directory object is created and recorded in a logical container specified as a command line argument by the user, e.g.

```
  %  wbserver -reset 1
```

stores a directory object in logical container 1. This dictionary object is used to maintain mappings for current whiteboards[12] It is consulted on a warm start, so that the whiteboards it knows about may be accessed.

2.3.2.1 Hash, Buffer and Directory Objects

The directory stores the whiteboards in the system, and maps a text name to a whiteboard. This mapping is accomplished using a hash object, and the directory contains an array of hash objects. The definition of hash is:

```
//
// hash Definition
//
permclass hash{
           whiteboard *wb;
           char name[NAMESIZE];
public:
           global Zero(){wb = 0;}
                          // empty the hash object

           global int IsZero(){return(wb == 0);}
                          // is the hash object empty?

           global Assign(whiteboard *newwb, const char *newname);
                          // assign a whiteboard and name to the hash object

           global int IsSame(const char *newname);
                          // is the hash object the same as another with newname

           global whiteboard *GetWB(){return wb;}
                          //return a whiteboard

};
```

[12] The current implementation does *not* use `lookup` and `record`, but instead implements its own directory object. It could be changed to do so.

The hash object is considered to be empty if the `wb` reference is zero. Two hash objects are considered to be identical if the names are the same.

When the names of the whiteboards in the directory are printed in `Contents` (see below) a `bufferType` object is used. The `bufferType` object is defined as follows:

```
//
// bufferType Declaration
//
permclass bufferType{

           char thebuffer[(DIRSIZE*NAMESIZE) + 1];

public:

           global void  NewBuf(const char *newbuffer);
                         // copy the contents of newbuffer into thebuffer

           global void OutputNames();
                         // output the contents of thebuffer

};
```

The directory class is defined as follows:

```
//
// Directory Declaration
//
permclass Directory{

           hash table[DIRSIZE];

public:

           Directory();
                         // constructor

           global Register(whiteboard *wb,const char * name);
                         // enter a whiteboard into the directory

           global whiteboard *LookUp(const char  * name);
                         // lookup a whiteboard using its name

           global Remove(const char * oldname);
                         // remove a whiteboard from the directory

           global Contents(bufferType *allnames);
                         // show the contents of the directory

           global CleanUp(screen *old);
                         // clean up the directory by removing any
                         // references to old screens
};
```

When creating a whiteboard a name for the whiteboard (a string) and a logical container (a low valued integer) must be specified in which to store the new whiteboard, allowing the distribution of the storage for the whiteboards over several nodes. A whiteboard may be looked up by specifying its name. `Contents` lists the names of the whiteboards in the directory and `CleanUp` removes all references that other objects in the application may have to an old screen (see below).

2.3.2.2 Screen

A screen object is created locally when the application is warm started, and conceptually it models a terminal screen. The screen object contains a reference to the directory object so that it may access whiteboards, and a reference to the whiteboard currently being viewed, if any. The declaration of the screen class is:

```
//
// Screen Declaration
//
permclass screen{

           whiteboard *currwb;  // the current whiteboard that the screen has
                                            // selected
           Directory *dir;  // a reference to the recorded directory object

public:
           screen();
                    // constructor

           global CreateWB(const char *name,int contno);
                    // create a whiteboard specifying its name and the logical
                    // container number in which it is to be placed

           global SelectWB(const char *name);
                    // select (or view) a whiteboard

           global ChangeWB(const char *name);
                    // select a different whiteboard to the one previously
                    // selected

           global DeleteWB(const char *name);
                    // remove a whiteboard from the directory

           global EnterText(const char *name, const char *text);
                    // enter text into a whiteboard replacing the previous text
                    // if any

           global ListWB();
                    // list the names of the whiteboards in the directory
           global Update(const char *text);
                    // update notifies all screen objects
                    // that have selected it that its text has changed

};
```

The screen object may create and delete whiteboards, as well as entering or changing the text of a whiteboard. When a screen selects a whiteboard, it is informed if the whiteboard's text is changed by another screen object running on the same or a different node.

2.3.2.3 Class Whiteboard

A whiteboard object consists of a text name (e.g. "wb1"), a text word of "information", and an array of references to screens. The definition of `whiteboard` is:

```
//
// Whiteboard Definition
//
permclass whiteboard {

           char name[20];  // the name of the whiteboard
           screen *displayedon[MAXSCREENS];  // the screens that have selected
                                             // this whiteboard
           char info[20];           //the text of the whiteboard

public:

           whiteboard(const char *name);
                         // constructor

           global MakeCurrentScreen(screen *newscreen);
                         // add newscreen to the displayedon array

           global RemoveCurrentScreen(screen *oldscreen);
                         // remove newscreen from the displayedon array

           global AddText(const char *text);
                         // add new text to the whiteboard

           global GetText(bufferType *newbuf);
                         // write the text of the whiteboard into newbuf

           global const char *GetName();
                         // get the name of the whiteboard

           global CleanUp(screen *old);
                         // remove the old screen reference from
 //  the displayedon array

};
```

The screen references refer to screen objects that have selected that whiteboard at that time. When the whiteboard's text changes, it checks the array, and for every screen reference found, it sends the screen a new copy of its "information". Whiteboards are persistent and screens are not, so when the application exits, the reference to the screen object that no longer exists must be removed from the array so that when the whiteboard's text changes in the future it will not try to send an update to that non-existent screen. This is done by the `CleanUp` method.

2.3.2.4 I/O Object and Mainline

The purpose of the I/O object is to provide a menu to the user so that whiteboards may be created, viewed and changed. The I/O object is created locally with the screen object when the application is started on a warm start and it has a reference to the local screen. The I/O object is declared as follows:

```
//
// IO Declaration
//
permclass IO{

           screen *myscreen;  // the local screen object

public:

           IO(screen *newscreen);
                        // constructor

           global Run();
                        // the menu

};
```

The constructor initialises the `myscreen` reference to `newscreen` and the `Run` method provides the menu.

The mainline creates a new directory on a cold start and places it in the logical container specified by the user. On a warm start, a screen and I/O object is specified and the menu is run.

Chapter 3

C** specifics

3.1 Persistent classes

C** extends C++ by allowing specified classes to support persistent instances.

Not all instances of a persistent class necessarily persist. When a program exits, and potentially at other times, only those instances of a persistent class which are (transitively) reachable from a Name Store (c.f. section ??) are themselves considered persistent. However the same compiled code for a class can be executed on both persistent and non-persistent objects from that class.

C** introduces two new class categories to facilitate persistence, in addition to the usual `class`, `struct` and `union`.

Any `class` which inherits, directly or indirectly, from a `permclass` can also support potentially persistent objects. Any class which supports potentially persistent instances is called a "persistent class". Objects from a persistent class persist if they are dynamically allocated using `new`, and they are (transitively) reachable from a Name Store.

A `volclass` is in all respects identical to a C++ class: in particular its instances are *never* persistent. A `permclass` supports *potentially* persistent instances – potentially because of (transitive) reachability from the Name Store, as in the manner described above.

Note that although a `volclass` is equivalent to a C++ `class`, it is `not` in general equivalent to a C** `class`: a C** `class` will be persistent if it inherits from a `permclass` – a `volclass` is *never* persistent, even if it does inherit from a `permclass`.

Each persistent class so identified by the compiler is associated with a compiler generated class, the so called *upcall* class for that persistent class. When an object of a persistent class is created using `new`, an upcall object is physically attached to the new object by the storage management routine[1]. Each heap allocated object of a persistent class has just a single associated upcall object: that is, member objects of an object of a persistent class, which themselves are from persistent classes, share the same common upcall object:

```
permclass LogBook {
   ...
};

permclass Car {
```

[1]ie _user_new is overloaded by the compiler.

```
    ...
    LogBook MaintenanceRecord;
    ..
  };

  ...

  Car *c = new Car();             // single upcall object for Car
                                  // and MaintenanceRecord
```

The member functions of an upcall class are automatically generated during compilation, and include identifying the location of all pointers and references in the instance data. This information is then used at execution time to ensure that the entire graph of connected persistent objects is faithfully stored and restored to and from storage.

When a pointer to a persistent object is dereferenced, and the target object is not currently resident in memory, then some overheads are incurred while the target object is being fetched. Subsequently, *all uses of that pointer incur no additional overheads beyond that of C++: the pointer refers directly to the target object in the usual way* – even if it is a nested member such as `MaintenanceRecord` above.

As a side effect of fetching one object from the store, the Amadeus runtime may fetch further objects (in the same "cluster"), thus reducing the cost of subsequent "object faults" on the prefetched objects.

3.2 Global classes

A class which has a global member function is termed a "global class". Further, any class which has member objects of global classes is also global, whether or not it itself has any global member functions. The global functions of an instance of a global class are termed the "global features" of that class. A class can also have a global class as a member of that class: in this case the global features of the (enclosing) class include those of the member. For example:

```
  permclass LogBook {
    ...
    global void MakeEntry (const char*);  // make entry in log book
  };

  permclass Car {
    ...
    LogBook MaintenanceRecord;
    ..
  };
```

Here `Car` is a global class, since it contains a nested member `MaintenanceRecord` which is global because it contains a global member function.

The global features of an object of a global class can be accessed via a pointer, independently of the current (node) location of that object. As noted above, these global features may in fact be nested parts of a global class.

Thus, given the above declarations, the following (remote) call can be made from a remote node:

```
Car *c;
....
c->MaintenanceRecord.MakeEntry('Serviced on 16-12-91');
...
LogBook *lb = c->MaintenanceRecord;
...
lb->MakeEntry('Serviced again on 17-12-91');
```

Global features can be inherited in the usual manner of C++, and can appear in the `public`, `protected` or `private` interfaces in the usual way. For example:

```
permclass LogBook {
  ...
  global void MakeEntry (const char*);  // make entry in log book
};

permclass Car {
   ...
 private:
   LogBook MaintenanceRecord;
   ..
};
```

Thus, the following calls can still be made from a remote node:

```
Car *c;
....
c->MaintenanceRecord.MakeEntry('Serviced yet again on 18-12-91');
...
LogBook *lb = c->MaintenanceRecord;
...
lb->MakeEntry('Serviced finally on 19-12-91');
```

Naturally, by the rules of C++ for `private` members, such a call could now only be made within a member function or friend of class `Car`: for example, by a peer object (of class `Car`). Nevertheless, in C** such a call can be made (e.g. by a peer object) across the underlying network.

Global member functions are always (implicitly) `virtual`, and thus a redefinition in a derived class overrides that in the base class in the usual manner.

Instances of a global class can potentially migrate from one node of the distributed system to another. Thus access to a global feature is always *potentially* a remote access: even if the target global object is currently local, there is no guarantee that it will remain so in the future. Remote access is only supported for global features. Given a pointer to a global object, the current definition of C** restricts access to that object to its global features: there is no access to the non-global features even should the designated object actually be local. Consequently, objects of global classes are totally encapsulated when accessed via pointers: not even peer instances of the same class or friends can access any of their private, protected or public members other than those member functions which are explicitly marked as global and those member objects which are themselves global.

In Amadeus v2.1, member by member copying of objects of global classes is not supported - thus it appears as if the default constructor of each global class is private.

When a global object is migrated, some of the non-global objects it (transitively) references are migrated along with it. Those non-global objects which are instances of `permclass`es are migrated with it, while all other objects remain behind. Global objects can obtain notification of migration events by defining `unmap` and `map` member functions (c.f. section ??), and so rebuild pointers to non-global, non-persistent classes.

In C**, non-global objects are never shared[2] between global objects. Each global object forms the root of a subgraph of non global objects. In order to preserve this assumption, pointers to non-global objects are never transmissable as arguments or results of global member function calls.

3.3 Marshalling

Each global member function is supported by an RPC stub so as to allow remote use of the object via a pointer. However, there are a number of limitations imposed on the type of arguments and results which can be marshalled. In C** v2.0, the types which could be marshalled were:

1. pointers or references to a global class
2. value (`const` or otherwise) of a basic type: `int, float, char, unsigned int`
3. value (`const` or otherwise) of an `enum` type
4. `const char*`, for which a null terminated byte string is expected
5. manifestly sized arrays of any of (1) to (5)
6. `struct`s whose fields are any of (1) to (6)

Notably arrays whose sizes are determined at runtime, and member by member values of classes cannot be passed in the present C** implementation.

Further, pointers to basic values (such as ints) are also currently not supported, other than `const char*`. This is because the only globally referencable entities in C** are global objects, and an `int` is not an object. Naturally a programmer may define her own encapsulation of e.g. an `int` as a global class if cross-machine pointers to `int`s is really desirable.

We intend to increase the range of supported types in the future. We are for example considering adding support to allow for in/out transmission of simple types such as `int`s, although not necessarily by presenting these to the programmer as pointers to `int`s.

3.3.1 Additions in v2.1

In addition to the above list, we have added the ability to pass and return objects-by-value. The details of this are summarized below:

1. the marshalled objects may contain any of the (1) to (6) above
2. pointers to temporary objects or basic types are re-initialised to zero (as they are no longer valid in a different context), *except* if the `-norp` command line switch is enabled, in which case they are not re-initialised.
3. references to temporary objects cannot be reset to 'valid' values, and a warning is issued to that effect
4. while arrays of pointers and values can be handled, arrays of objects cannot yet be processed.

[2]Again, with the corollary that type casting by the programmer can circumvent this.

An example of an object being marshalled by value is given in amadeus/demos/manual_examples/marshal, and an extract is given here :

```
class param : public glob_cl { // this is marshalled recursively
public:
   int i = 8888; // is marshalled

   int *ip;      // this is a pointer to a non-global value; it is be
                 // reset to zero
                 // (unless the '-norp' switch is set)

   int &ir;      // references to temporaries are invalid when marshalled, and
                 // cannot be reset; a warning to this effect is issued

   char *cp = "01234567890";                 // as in 'ip'
   const char *ccp = "abcdefghijklmnop";     // is marshalled

   temp_cl t;          // is marshalled by value recursively
   temp_cl *tp;        // as in 'ip'
   temp_cl &tr;        // as in 'ir'

   glob_cl g;          // is marshalled by value recursively
   glob_cl *gp;        // is marshalled, will still point to its global object
   glob_cl &gr;        // is marshalled, will still reference its global object

   int iarr[2];        // is marshalled by value recursively
   int *iparr[2];      // as in 'ip'
/*   temp_cl tarr[2];  // error - unfortunately, arrays of objects cannot be
                       // marshalled by value yet
*/
   temp_cl *tparr[2];  // as in 'ip'

/* glob_cl garr[2];    // as in 'tarr' */
   glob_cl *gparr[2];  // as in 'gp'
...
}
```

3.4 Amadeus events

Editorial Note:
This feature is not supported in the publicly released C** v2.1. It is considered a high priority.

Each persistent or global class can *optionally* have its own member functions to respond to particular events occurring in the Amadeus environment. These currently are:

`void unmap()` called when the current object is about to be unmapped from memory and stored onto disk or migrated.

`void map()` called when the current object is fetched from disk or from another node and brought into memory. This call might be used to rebuild pointers to objects of volatile classes.

3.5 Exceptions

Although g++, on which the current C** implementation is based, provides support for exceptions, C** does not provide exception support, pending the implementation of the exception model described in the Annotated Reference Manual.

As an interim measure, C** allows the application programmer to specify a function - a so-called *environment exceptions handler* - to be called whenever an exception is raised in the underlying Amadeus environment.

The programmer can specify one environment exception handler per *activity* (see below in section ??). This function is called in the context of the activity, when various unusual conditions arise during the execution of the activity.

To install a new environment exceptions handler use:

```
handler_func set_eehandler(handler_func Handler);
```

where `Handler` has the form:

```
extern void (*foobar (int)) ();
```

where `foobar` is the name of the function. The parameter passed to the handler is the *exception identifier* which indicates which of the set of predefined exceptions has occurred.

`set_eehandler` returns the previous handler function for the activity so that the programmer can implement a stack strategy for environment exceptions handlers. Each activity initially has a default environment exceptions handler which will terminate the activity having first printed an appropriate error message.

3.6 Concurrency

Amadeus provides a concurrency model based on the concepts of *activity* and *job*.

A *job* is at first sight similar to a classical Unix process. The difference from a classical Unix process is that a job can span multiple nodes. In particular, a call to a member function of a remote object can result in the current job executing that member function remotely at another node. When this happens for the first time, the job is said to have *diffused* to that node.

An *activity* is likewise similar at first sight to a classical lightweight process or thread. A job may contain one or more activities, running in parallel[3]. An activity can however diffuse to another node as a part of its job, thus spanning multiple nodes.

Below we first present the general design for management of activities and jobs in C** . Unfortunately this design is not yet implemented: and so instead we consider in section ?? how a similar result can be achieved by explicit hand coding. Finally in section ?? we present some macros which assist in C** v2.1.

[3]or, at least, quasi-parallel in Amadeus v2.1.

3.6.1 Futures

In C**, this model is presented through the concept of *futures*, as appeared in "Multilisp" by R. Halstead. Instead of invoking a member function synchronously, and waiting for its result, a new job or activity can be "forked" off to execute the member function in parallel to the caller. At some later stage the caller can test for termination of the forked activity or job, and recover the results of the member function call. Equally, the caller may choose to ignore those results and never synchronise with the termination.

Member functions which can be used for futures must be explicitly marked using the keyword `active`: `active` member functions are also treated as `global`.

For example, given:

```
   class foo {                                    // or permclass, or volclass..
        ...
        active result fn (args);
        ...
    }
```

the function `fn` can be used as the initial operation of a new job or activity.

A new job or activity can then be created and controlled as follows:

```
#include <amadeus.h>

    foo* f;

    f = new foo ();               // create a foo

    result r = f->fn (args);      // foo::fn() can be used as usual

    future *ft = f->fn (JOB, args);// create a new job

    ft->suspend();                // suspend the job

    ft->resume();                 // resume the job again

    if (ft->done())               // can we get the result yet?
         {
          f->redeem(ft, r);       // pick up result of forked call here...
         }
```

Note that the `active` member function of `foo` can still be used as normal. `fn` can also be called to create a new job or activity in which case it takes an extra parameter (of type `actjob_t`) to specify whether a job or activity is required and returns a (pointer to a) future to be used to control the job or activity subsequently. The job/activity can be suspended or resumed as often as is required. Moreover the new job or activity can be explicitly terminated by means of the `kill` function.

Each `active` member function is expanded by the C** compiler into two functions: one to create a future, and one to obtain its result.

3.6.2 Futures in C** v2.1

This section shows by example how jobs and activities can be created in C** v2.1.

3.6.2.1 Hand-coding

As a first example, consider the following class definition:

```
class foo {

public:
    global void launch(int a, ref *b, int c);  // operation no. 1
                                                // ref is a global class

};
```

Only operations marked as `global` may be used for creating jobs or activities. The operation numbers (see below) are assigned to all global operations of a class consecutively starting from 1 in the order in which the operations are declared. To create a job or activity to invoke the operation `foo::launch` on an instance of `foo`, another C** method is required. This method should be declared as follows:

```
#include <amadeus.h>
#include <future.h>

class bar {

public:

    future* future_launch( foo* f, actjob_t which,
                           int a, ref *b, int c);

};
```

The first parameter `f` is a pointer to an instance of `foo`. The parameter `which` specifies whether a job or activity is to be created (`ACT` specifies an activity and `JOB` specifies a job). The necessary definitions may be obtained by including the file `amadeus.h`. The remaining parameters should match the ones given in the declaration of `foo::launch`.

Class `future` is defined in the include file `future.h`.

The body of `future_launch` should be as follows:

```
future_launch( foo* f, actjob_t which, int a, ref *b, int c)
{
    // change these next four line accordingly
    const int    opid  =  1;
    const int    dsize =  sizeof(a)+sizeof(c);
    const int    nrefs =  1;
    const aon_oo *obj  =  (aon_oo *) f;

    future *ft = new future(which);

    aon_stub s1;
    amadeus.getstub (ft, s1);

14:     aon_oo* _t = obj->object();
15:     aon_marshal m (_t,  opid, dsize, nrefs, which, s1, s1);
16:
17:     // push all parameters
18:     m.push(a);
19:     m.push(b);
20:     m.push(c);
21:
22:     void *amh =  (void*) ((int)m.block() - 236);
23:
24:     aon_absentcluster (amh, m.badd());
25:
26:     return ft;
27: }
```

`Opid` (line 4) should be changed as appropriate according to which operation is being invoked. The constant `dsize`, on line 5, is the size of the parameters *excluding* any object references. The number of object references is given next in `nrefs` (line 6). Here, there is one object reference in the parameter list, so this is given as 1. The `f` in line 7 refers to the object being invoked. Lines 18 to 20 should be changed depending on the parameters to the operation. Parameters should be pushed in the order that they are declared. Note that *only* lines 4-6 and 18-20 need be changed, as described, to create a job/activity to invoke a different operation.

Note that if `future_launch` is a member function of class `foo`, the first parameter `f` can be dropped and the reference to `f` in line 6 should be changed to `this`.

The code necessary to actually create a job/activity is as follows:

```
1: {
2:
3:     future *ft;
4:     foo *f = new foo;
5:     bar *b = new bar;
6:
7:     ft = b->future_launch(f, ACT, 1234, b, 4567);
8:
9: }
```

This example creates an activity (`ACT` on line 7). Note that here, instances of `foo` and `bar` are explicitly created. They could, however, have been created elsewhere.

3.6.2.2 Using macros

The second example illustrates the use of macros which hide and simplify the implementation of futures. The macros we have defined in the release of C** v2.1 may only be used to implement futures of methods of a class if those methods take a single integer parameter. However the macros can of course be edited and refined if required.

Consider a modified version of class `foo`:

```
1: class foo {
2:
```

```
public:
    global void act0(int p);
    global void launch1();
    global void launch2();


};
```

Class `foo` has three global functions, the *first* of which must take a single integer parameter. To implement futures for `launch1` and `launch2`, class `bar`, as introduced above in section ?? is unnecessary. In its place two macros are used.

The first macro, `ACTIVE`, takes a class name as a parameter:

```
ACTIVE(foo)
```

and is used to indicate an *active* function, which is *always* the first global function in the class, in this case `act0`. To use these macros `act0` *must* take a single integer parameter and not return any results. The purpose of `act0` is to multiplex the other functions in the class that are to be invoked using a future. For example, the implementation of `act0` here is:

```
void foo::act0(int p)
{
  switch(p){
      case 1: launch1();break;
      case 2: launch2();break;
    }
}
```

If `p` is 1 then `launch1` is called, and correspondingly if `p` is 2, `launch2` is called.

The second macro, `BUILD_ACTIVATE`, also takes the class name as a parameter:

```
BUILD_ACTIVATE(foo);
```

and is used to implement the future support for the active function (`act0`) of the specified class (`foo`). The support provided is essentially that of the code in `future_launch` in section ??.

The code to create a job or activity is called using the macro `ACTIVATE`, which takes three parameters:

```
ACTIVATE(CLASS *c, actjob_t which, int select);
```

`c` is a reference to an object of type `CLASS` (in this case `foo`); the parameter `which` denotes an activity or job as before; and `select` is used by the active function `act0` to determine which function (`launch1` or `launch2`) to call. The code to create an activity for `launch1` and `launch2` is:

```
{

    future *ft1,*ft2;
```

```
 4:      foo *f = new foo;
 5:
 6:      ft1 = activate(f,ACT,1);
 7:      ft2 = activate(f,ACT,2);
 8:
 9:  }
```

An activity to execute `launch1` is created on line 6, and another for `launch2` on line 7.

3.7 Synchronisation

As stated above, the Amadeus activity is analagous to the concept of a process thread. Programmers wanting to use threads or activities need to take special care when sharing data between threads or when using Unix system calls.

Sharing data between threads simply requires synchronised access to the data, as outlined below. Standard Unix libraries such as *clib* do not currently support multi-threaded execution. A number of difficulties arise in the areas of non re-entrant system calls, the errno variable, signals and blocking I/O[4]. Current thread packages such as *Cthreads* or Sun's *lwp* package offer no real library support for threaded programming. The *Pthreads* initiative, as part of the POSIX standard intends solving all related problems.

The following gives examples of how Amadeus supports use of activities in view of the problems outlined above.

3.7.1 Shared data areas for activities.

Amadeus provides the *semaphore class*, which can be used to synchronise access to shared data. The definition of the semaphore class is as follows:

```
permclass semaphore : public aon_semaphore {
public:
           semaphore():(1) {}                 // binary semaphore
           semaphore(int i):(i) {}            // general semaphore

           global void wait();                // wait
           global void condwait();            // conditional wait
           global void signal();              // signal
           };
```

where the usual `wait, signal` and `condwait` operations are defined. The semaphore class can be used by including the file `sem.h`.

To synchronise access to shared data a user program must `wait` on a semaphore before accessing shared data and `signal` the semaphore afterwards. Generally there is one semaphore per shared data section, however the programmer may opt for a finer granularity[5].

[4]For a complete discussion of the issues see *Bringing the C libraries with us into a mutli-threaded future* by Jones, Usenix 1991.

[5]i.e. a semaphore per field of a shared data area so as to improve concurrency.

The following is an example of how two activities may synchronise access to a shared integer.

```
// Shared semaphore
semaphore *mutex;

//
// Activity One
//
while(1){
        mutex->wait();
        shared_int++;
        mutex->signal();
}

//
// Activity Two
//
while(1){
        mutex->wait();
        shared_int++;
        mutex->signal();
}

// Extract from mainline
.
.
mutex = new semaphore();
```

3.7.2 Making system or library calls

Difficulties arise when multi-threaded programs use the standard Unix libraries due to the following reasons:

- Some functions (e.g. *getserverbyname*) are non re-entrant as they return a pointer to a data area which is allocated on a *per process* basis.
- Some functions maintain state between calls (e.g. *malloc* or file access) and suffer from the same problems as non re-entrant functions
- The Unix *errno* variable is allocated on a per process basis.

The C** programmer is provided with basic support for the above problems, namely the ability to *serialize* system calls. The user is allowed to define *critical sections* of application code, which cannot be interrupted. Amadeus guarantees that an activity which is in a critical section, will not be preempted until it exits the critical section, therefore guaranteeing an activity unobstructed *use* of a library call.

The following is an example of an activity which makes a system call inside a critical section, accessing (or possibly copying) the results before leaving the section.

```
// Enter a critical section
amadeus.enter_critical();

cout << "Activity is non-preemptable\n\n";

hostent_ptr = getserverbyname("HOST");

// Now read required field from per-process buffer
// with guarantee of non-preemption


// Finally leave critical section and allow premption
amadeus.exit_critical();
```

3.7.3 Blocking I/O system calls.

Many standard output functions (e.g. `printf,write`) are asynchronous semantically. However most standard input functions (e.g. `read, scanf`) are synchronous, meaning that they block until input is available. This may be unacceptable to the threads programmer, as a call to a blocking I/O function will block the *entire* process of threads. It is usually desirable to only block the calling thread.

Amadeus provides facilities whereby an activity can *await* input or output on a specified file descriptor. This facility allows other threads to be scheduled while the thread is awaiting input or output. The following is an example of how a thread may await input on `stdin`[6].

```
char* string;
int r = 1; // r = 1 => await input on descriptor.
int w = 0; // w = 1 => await output on descriptor.

int i = amadeus.await_fd(0,&r,&w);

if (r){
        cout << "******RECEIVED INPUT*******\n";

        // Enter a critical section (No preemption)
        amadeus.enter_critical();
        scanf("%s",string);
        amadeus.exit_critical();
}
```

In the case where more than one thread is awaiting input or output on a common file descriptor, i.e. the file descriptor is a shared variable, then the programmer must synchronise access to the file descriptor using semaphores as explained previously. In the above case, this would involve encapsulating the entire piece of code inside a semaphore.

[6]Recall that file descriptor 0 is `stdin`.

3.8 Using Signals.

Currently Amadeus provides no extra support for Unix signals in a threaded environment. Signals are delivered and handled on a per process basis.

3.9 Managing storage

The distributed store of an Amadeus installation is divided into a number of *containers*. Above Unix, each container is implemented as a directory containing a number of files. Within C** programs, containers are identified by system-wide unique `int`s.

Each container can store up to 2^{24} persistent objects. Each such persistent object itself forms the root of a subgraph of objects (c.f. section ??). Groups of objects can be stored together in *clusters*, in such a way that an "object fault" which brings in an object will also fetch from store the other objects belonging the same cluster.

In Amadeus v2.1, there is no direct programmable control over clusters from a C** program. Allocation of objects to clusters is done when the objects are created, and cannot be changed thereafter.

A C** programmer can establish which container should be used to store all subsequently created `permclass` objects, by using the Amadeus library call `setlc`. Naturally only those `permclass` objects which are not manifestly garbage are actually stored. `setlc` can be used at any time to change the current container used for newly created objects.

The Amadeus library provides further calls to create a container, mount and unmount a container, and obtain the associated Unix directory name for a container. Chapter ?? describes these in more detail.

Chapter 4

The Amadeus runtime

The set of down-calls described below can be invoked on the Amadeus runtime from a C** program.

```
int               reset();
void              record(const char *Unixfilename,  void *object);
void*             lookup(const char *Unixfilename);
int               getnodeid();
int               setlc(int LcNum);
int               getlc();
int               createlc(char* LcPath);
int               wherelc(int LcNum, char* LcPath);
int               mountlc(int LcNum, char* LcPath);
int               unmountlc(int LcNum);
handler_func      set_eehandler(handler_func Handler);
int               getpid(void*);
```

reset() returns true if the `-reset` flag was passed to the application.

record() registers the specified object in the Name Store under the specified name.

lookup() returns a pointer to the object registered with the specified name in the Name Store.

getnodeid() returns the identifier of the local node.

setlc() sets a given container to be the "current" container. The given container has to be already mounted on one of the nodes.

The parameter is: **int LcNum**, the container's identifier.

Return value: 1, if successful; otherwise 0.

getlc() obtains the "current" container identifier.

Return value: the current container identifier.

createlc() creates a new container. It allocates a new container identifier, and creates the control files associated with a container. This function does not mount the new container.

Parameter is: **char* LcPath**, the container's directory path.

Return value: The new container number, or zero on error.

wherelc() searches the local and remote mount tables for a specified container. If found, the node number of the control node, and optionally the container's path are returned.

Parameters are:

1. **int LcNum**, the number of the container being searched for.
2. **char* LcPath**, the container's path. This argument is ignored if it is empty.

Return value: The node number of the container's control node, or zero if it is not mounted anywhere.

mountlc() mounts a container on the current node. It checks that the container is not mounted elsewhere already, and if not, a message is sent to other servers telling them that the container is mounted on this node.

Parameters are:

1. **int LcNum**, number of the container being mounted.
2. **char* LcPath**, path of the container's directory.

Return value: 1 on success, 0 otherwise.

unmount() makes the given container inaccessible. There should be no objects from the given container mapped in memory when this down-call is issued.

The only parameter is **LcNum**, number of the container being unmounted.

Return value: 1 on success, 0 otherwise.

set_eehandler() sets the environment exceptions handler for the current (calling) activity.

Return value: the previous environment exceptions handler for the activity.

getpid(void)* given a C** pointer, returns a short unique identifier for the object (its pid - a logical container number and a unique number within this).

Appendix A

Summary of restrictions in the C** v2.1 implementation

As noted in the text, the current C** implementation imposes a number of undesirable restrictions, which we hope to remove in the future. These are summarised here for convenience:

- restrictions on marshalling
- the Amadeus environment does not yet allow global, but non persistent, classes. Currently therefore C** converts such declarations into global and persistent classes, and provides a warning that it is doing so.
- `active` member functions are not yet compiler supported.
- `unmap` and `map` are not yet supported.
- Exception handling is not currently supported.
- `unions` as members (data) of `permclass`es are not yet supported.
- An object cannot be migrated between heterogeneous hosts.
- `static` data (such as `<stream.h>`'s `cout`) are not accessed correctly in inline member functions in a class definition (a bug!).
- Explicit use of `delete` on a `permclass` object fails to interact correctly with the Amadeus garbage collector (a bug!).
- anonymous `struct`s and `union`s are not correctly handled (a bug!).