An Extensible Dynamic Linker for C++

by

Murali Krishna Vemulapati

Submitted to the Department of Civil and Environmental Engineering
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author .... [illegible]
Department of Civil and Environmental Engineering
May 15, 1995

Certified by .... [illegible]
Ram Duvvuru Sriram
Associate Professor
Thesis Supervisor

Accepted by .... [illegible]
Joseph M. Sussman
Chairman, Departmental Committee on Graduate Studies

An Extensible Dynamic Linker for C++

by
Murali Krishna Vemulapati

Submitted to the Department of Civil and Environmental Engineering
on May 15, 1995, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

In this thesis, I have designed and implemented a portable and extensible dynamic linker for C++. The dynamic linker, *Dld++*, allows a user process to dynamically link compiled C++ code and load it into its address space; it also allows the process to unlink such dynamically linked object code and unload it from its adress space. *Dld++* performs at run-time those tasks which are performed by traditional linkers at link time. These tasks include loading of simple object files, searching library archives, linking against shared libraries, relocating text and data, and resolving symbol references. *Dld++* also collects special symbols from an object module into lists and performs user-specified actions on these lists. For example, for every object file that is dynamically linked, *Dld++* collects all the symbols corresponding to the global/static constructors and invokes those constructors. *Dld++* uses the general purpose Binary File Descriptor (BFD) libraries of Project GNU to operate on the object files which makes the linker portable across several object file formats. The dynamic linker itself is implemented as a library of C++ classes so that it is easily extensible. I demonstrate with examples how *Dld++* can be modified or extended for different C++ compilers and operating system environments.

Thesis Supervisor: Ram Duvvuru Sriram
Title: Associate Professor

Acknowledgments

I would like to express my gratitude to my thesis advisor Dr. Duvvuru Sriram for his guidance, advice and encouragement. He initiated me into the exciting field of object-oriented databases and software systems and it was he who first suggested to me the problem of dynamic linking in C++.

I would like to take this opportunity to thank my faculty advisor Professor Robert D. Logcher for his advice and guidance.

I would also like to thank Dr. Amar Gupta, of Sloan School of Management, for his advice and support.

Contents

List of Figures

Chapter 1

Introduction

Dynamic linking and loading refer to the ability to add new code to an already running program so that the new code can be accessed from within the old code. Traditional compile, load and go techniques require the shutting down of an application whenever a new software component has to be added to the application. It can be useful to allow a user to add a new software component to a running program on the fly so as to facilitate rapid prototyping.

Traditional linkers combine a number of separately generated object modules and archive library files, relocate their text and data to fixed virtual memory addresses, tie up their symbol references and construct a final executable image. Such an executable image is said to be *statically linked.* After an executable image is generated by static linking, it is loaded into a process by mapping the image to the address space of the process. Thus the programs are static entities in the sense that the construction of a program is completed before its execution. It is not possible to change or enhance the functionality of a program during its execution. Also, the entire program needs to be relinked whenever any of the object modules is modified or a new object module is to be added. The relinking operation is typically very expensive even when only a few object files are modified.

Dynamic linking, on the other hand, provides the ability for a process to add object modules to its address space or remove object modules from its address space at run time. In addition to providing considerable flexibility to a programming environment, dynamic linking provides several other advantages. Dynamically linked program images are significantly smaller on the secondary storage than the statically linked images. This is because commonly used libraries are not linked statically to the program images, but instead are linked dynamically to the process in the memory. Also, the static linking time is considerably reduced thus facilitating rapid program evolution. It also possible to reduce the size of the program images in the memory by letting several processes share a single copy of commonly used library routines.

Dynamic linking also has certain drawbacks associated with it. For example, a dynamically linked program (*i.e* an incompletely linked executable which needs further link editing at program start up time) needs to complete the link editing of undefined external symbols and hence takes longer to start the program each time. Also, the type-safety is compromised at the interface between the old code and the dynamically linked code. (The type-safety problem can be alleviated to a certain extent in C++ as we shall see later).

1.1 Previous Work

Dynamic linking was originally part of the operating system MULTICS [7], but it required the dynamic linking to be performed only in the kernel mode. This feature was not initially very popular as it introduced excessive overhead into program loading. In some of the later operating systems, dynamic linking was reintroduced as part of the shared library implementation mechanism.

1.1.1 Dynamic Linking and Shared Libraries

The UNIX operating system provides a facility whereby a single copy of program code in the physical memory can be shared among all the processes that execute that program. This greatly reduces the memory usage. However, most of the program executables also contain code for some commonly used library routines. For example, almost every C program will make use of standard C library function `printf`. Hence, every such program will have its own copy of `printf` function. It would result in a lot of saving in memory utilization if such common library routines can be shared between processes. Most modern UNIX operating systems [9, 14, 15] support such shared library mechanism.

Here, we briefly discuss about the shared library mechanism in SunOS [9]. The batch link editor in SunOS, *ld* combines a variety of module types such as object files, archives etc. to produce a final executable (typically known as an *a.out* file). In addition to `.o` files and `.a` files, *ld* also handles shared objects (`.so` files). A shared object is simply an executable without an entry point. When *ld* encounters such a shared object, it usually searches the file only for symbol information but does not include it in the final executable being produced. After *ld* processes all the modules, it typically creates an incompletely linked executable which needs further link editing at execution time because the shared libraries have not actually been linked to the executable. At program start-up, the program bootstrap routine *crt0* checks to see if the program needs dynamic linking of shared libraries. If so, it invokes the dynamic link editor, *ld.so* to complete the linking of shared libraries. The dynamic linker *ld.so* finds all the shared objects that were specified on the command line (that were not loaded yet) and loads them into the process's address space.

SunOS also provides a simple programmatic interface to its dynamic linker *ld.so* [20]. Operations are provided to add a new shared object to a program's address space, obtain the address bindings of symbols defined by such objects, and to remove such objects when their use is no longer required. The function `dlopen()` provides access to a shared object by loading that object into the address space and returning a descriptor to that object. The function `dlsym()` takes such a descriptor and a symbol name as arguments and returns the address of the symbol. The function `dlclose()` removes an object from the address space when the reference count for that object reaches 0. Even though, these functions can be used to provide simple dynamic linking capabilities to a program, they are not flexible. For one, no symbol table is maintained within the process and hence we can not perform dynamic linking in an incremental fashion. Further, we can only load shared objects; we cannot load either simple relocatable object files or archive library files.

1.1.2 The Dynamic Link/Unlink Editor *Dld*

Dld [10] is a dynamic linker that follows an approach for dynamic link or unlink editing based on a library of link editing functions that can add compiled object code to or remove such code from a process anytime during its execution. This is a *genuine* dynamic linker/unlinker in the sense that all the functionalities performed by a traditional linker, namely, loading modules, searching libraries, resolving external references, and allocating storage for global and static data structures, are all performed at run-time.

The basic functions provided by `Dld` are as given below:

1. The function **`dld_link`** dynamically links in the named relocatable object or library file into memory. If the named file is a relocatable object, it is completely loaded into memory. If it is a library file, only those modules defining an unresolved external symbol are loaded. Storage for the text and data of the dynamically linked modules is allocated in the heap of the executing process. After all modules are loaded, as many external references are resolved as possible.

2. The function **`dld_unlink_by_file`** is simply the reverse of the above function. The specified module is removed from the memory and the memory allocated to it is de-allocated. Additionally, the resolution of external symbols is undone.

3. The function **`dld_get_function`** returns the address of its string argument which should be the name of a function symbol.

Dld differs from other dynamic linkers in that it can not only add an object module to a running process, but also remove such object modules from the process. This facilitates modification of a program behavior at run time. We can simply unload a function definition and reload the function's new definition into the process. Even though *Dld* provides a very flexible approach to dynamic linking and unlinking, it still has some limitations.

1. It currently supports only one particular type of object file format known as the `a.out` format. So, *Dld* cannot be used on systems which support a different object file format. In order to port *Dld* to a different platform, we need to rewrite major portions of the source code.

2. It currently can handle only C programs. In the case of object modules from C++ programs, it cannot properly handle the static scope constructors and destructors.

3. It can not handle shared libraries. This means that it can not map shared objects into the memory. Also, if the main executable was dynamically linked (that is it made use of shared libraries when built), the dynamically linked modules cannot access the symbol information from those shared libraries. This necessitates building all the executables as statically linked.

4. The source language is C. Hence, the code is not directly reusable nor extensible.

In this thesis, we propose a new extensible dynamic linker *Dld++* which addresses these limitations of *Dld*. *Dld++* is a dynamic linker for C++ which is portable to several object file formats and is also easily extensible because it is designed and implemented using object-oriented programming concepts. The design centers around reusable *types* and new *types* can be built by extending the old ones. An object of such an user-defined type provides an encapsulation of a resource so that the resource can be used without knowing about its internal structure or state. Also, we can extend or modify the functionality of a resource by using the notion of *sub-typing*.

1.2 Incremental Linking in C++

Object-oriented design inherently supports incremental development of software. In the case of an object-oriented language like C++, the basic problem with incremental linking is to provide a technique for adding a new class to a running program [19]. To add a new class to a running program, we need to address the following issues:

1. We should be able to create instances of the class, which is yet to be defined, from the original program.

2. We should be able to invoke operations on objects of the new class from the original program in a type-safe manner.

Each of the problems has an elegant solution in C++ in the form of virtual functions. In the case of the second problem, in order to access the dynamically loaded code from a running program, the running program needs a late binding mechanism that maps the method calls to dynamically loaded methods. The virtual function dispatch mechanism of C++ is ideally suited for this. Each object has an embedded pointer to a per class virtual table of pointers to methods. A virtual function call on an object is implemented as an indirect call through the virtual table. Hence, as long as the newly loaded class is a derived class of a known class, it is possible to invoke operations on the instances of the newly loaded class in a type-safe manner.

A variety of solutions have been proposed under the generic label of "virtual constructors" for the first problem [8]. Even though C++ does not allow constructors to be virtual, a somewhat similar effect can be achieved by some simple programming techniques. For example, several techniques have been discussed for the virtual constructor idiom in [6]. One of them, which is of relevance to incremental code development, is the *exemplar idiom*. An exemplar of a class is a designated instance of that class from which new instances of the same class can be “cloned” by invoking a particular virtual function on it. A slightly different approach to this problem is presented in [2] in the context of an object-oriented operating system, *Choices*. It introduces a class *Class* to store information about newly defined classes. Each newly defined class has a *Class* object associated with it which contains a pointer to the addressable constructor of the class. The addressable constructor invokes the traditional constructor of the class.

1.2.1 Motivation for Dynamic Linking in C++

We see that incremental linking is desirable in certain applications and it is possible to incorporate such a facility into an object oriented system provided the services of a dynamic linker are available on the system. But when we add persistence to C++, we have to address a new problem . A program should be able to invoke operations on objects that are *already* existing on a persistent store. It may so happen that the program does not know the type of an object that it encounters on a persistent store because that instance was created by another program which shares the persistent store with the original program. Hence in the case of persistent languages, some form of dynamic loading of method code is necessary to handle such situations. In chapter 4, we will show how to build dynamic linking capability for a persistent C++ language called **E** by extending the dynamic linker *Dld++* and making some simple modifications to the run-time library support of **E**.

1.2.2 Issues in the Design of an Extensible Dynamic Linker

We have to address the following issues when we set out to design the dynamic linker *Dld++*.

1. **Portability** The dynamic linker should be portable across object-file formats. The design of *Dld++* makes use of the general purpose Binary File Descriptor Library (BFD) of Project GNU. BFD provides an abstraction of object file formats so that the applications can operate on object files without regard to their internal format.

2. **Compiler Dependencies** The dynamic linker should be usable with different versions of the C++ language. For example, each C++ compiler handles the initialiation of static scope objects in a different way. Since *Dld++* is designed in a object-oriented fashion, it is possible to modify or extend its behavior to suit the needs of a particular C++ environment.

3. **Operating System Dependencies** The dynamic linker should be able to handle the operating system (OS) dependencies. For example, the implementation of shared library mechanism differs from OS to OS. In this case, since the *Dld++* is easily extensible, it should be able to handle such dependencies. In chapter 3, we will show how *Dld++* can be extended to handle the shared libraries in SunOS.

4. **Extensibility** We should be able to add to the dynamic linker any other functionalities which a particular application might need.

1.3 Organisation of the Thesis

The rest of the thesis is organised as follows. In chapter 2, we discuss about the design and implementation of a basic version of *Dld++*. We briefly introduce the BFD libraries and present the implementation using the BFD libraries. In chapter 3, we will show how *Dld++* can be extended to handle static constructors/destructors and shared libraries. In chapter 4, we consider a full-length example. We will show how to build a dynamic linking capability for a object-oriented database language called **E**. Chapter 5 concludes the thesis.

Chapter 2

Design of *Dld++*

In this chapter, we discuss about the basic design and implementation of *Dld++*. *Dld++* is implemented as a library of C++ classes. This is basically a re-implementation of *Dld* [10] in C++ and using the BFD libraries.

2.1 An Overview of *Dld++* functionalities

The class `Dld` provides the basic functionalities of a dynamic linker. We start with creating a new `Dld` object. The constructor for class `Dld` takes the name of the executable file whose symbol table forms the basis for further incremental linking. The constructor performs the initialization of its internal data structures such as the symbol hash table. It further loads the external symbols from the executable file into its symbol hash table.

In order to dynamically link in a file into the process, we invoke the method `link` on the `Dld` object. The `link` method takes the name of the file to be linked and processes the object file just as a traditional linker does. If the file is a relocatable object file (so called `.o` files), it is loaded into the memory and relocated. If it is a library archive (`.a`) file, it is searched for those entries which match an unresolved external reference and those entries are extracted, loaded into memory and relocated. `link` searches library archives repeatedly until no more entries can loaded. The storage for both text and data (initialized and uninitialized) is allocated on the heap of the process.

In order to unlink an object module which has previously been dynamically linked into the process, we invoke the method `unlink` which takes the name of the object module to be unlinked. When an object module is unlinked, `unlink` undoes the resolution of references to external symbols defined by that module. It then reclaims the memory occupied by the text and data portions of the object module.

We illustrate the operation of *Dld++* with a simple example. Figure 2-1 shows a simple class `Base` with a virtual function `void describe()` and a class `Derived` which is derived from `Base` and redefines the virtual function `void describe()`. The source file `derived.c` which defines the `Derived` class contains an external data symbol `data_ptr` which is pointer of type `Base*`. The file also contains a global function `void create_instance()` which creates an instance of `Derived` on the heap and sets the pointer `base_ptr` to that instance. When the file `derived.c` is compiled by a C++ compiler, the resulting object module contains an undefined reference to the external symbol `_base_ptr`.

```
// File base.h

#include <iostream.h>
class Base {
  int x;
public:
  Base (int i) {x=i;}
  virtual void describe() {
    cout << "This is a Base class object." << endl;
  }
};

// File derived.c

#include "base.h"
class Derived :public Base {
public:
  Derived(int i):Base(i){}
  virtual void describe() {
    cout << "This is a Derived class object." << endl;
  }
};

extern Base* base_ptr;

void create_instance() {
  base_ptr = new Derived (0) ; // create a Derived object on the heap
};
```

Figure 2-1: `Base` and `Derived` classes with virtual functions

Figure 2-2 shows a main program which uses the *Dld++* functionalities. The figure shows the source file `sample.c` which is compiled into an executable named `sample`. We invoke the executable with one command line argument which is the name of the file to be dynamically linked into the process. (It is possible to read in the name of the file from standard input from within the program itself). In our example, the object file to be dynamically linked is `derived.o` and so we invoke the program thus:

```
sample derived.o
```

The main program creates a new `Dld` object `dyn_linker` on the heap and supplies the name of the executable (in this case `sample`) to its constructor. It then invokes the method `link` on `dyn_linker` with the argument (`derived.o`) to link in the object module into the current process. The dynamic linker loads the text and data sections of the object file into the heap of the process and relocates the text and data symbols. In particular, the undefined external symbol `_base_ptr` in `derived.o` gets resolved to the symbol defined in the main program. The main program then gets the address of the function symbol `create_instance__Fv` defined in the object module `derived.o` by invoking the method `get_function`. The method `get_function` gets the value of a function symbol by looking it up in the dynamic linker's symbol hash table. The function `create_instance` is invoked through the `func_ptr`. At this point, the pointer `base_ptr` points to a `Derived` instance. We invoke the virtual function `describe ()` on `base_ptr` which invokes the correct method (namely, `Derived::describe()`) and prints the following on the standard output.

```
This is a Derived class object.
```

Finally, the main program unlinks the object module `derived.o` from its address space by invoking the method `unlink`.

In summary, we have shown how a program can dynamically link a hitherto unknown derived class definition to a running process and invoke functions on the objects of the derived class. In particular, any virtual function invocation on an instance of a derived class is type-safe. In the next chapter, we will show how to create instances of a unknown derived class in a type-safe and general way. This necessitates some extensions to our basic *Dld++* which handle the static scope C++ objects in a module that is to be dynamically linked.

2.2 Designing with BFD Libraries

This section contains a brief introduction to using BFD libraries for building applications such as *Dld++*. A more detailed account can be found in [4]. We present a simple BFD-based application which illustrates several functionalities of the BFD library.

2.2.1 Overview

Binary **F**ile **D**escriptor (BFD) package is a library of routines which allows applications such as linkers operate on object files without regard to their internal format (such as a.out, elf, coff etc.). BFD provides an abstration of object files by providing a common interface to the object files. BFD has two major parts: a front end and a set of back ends one for each object file format.

1. The front end provides a common interface to the application programmer. It manages several canonical data structures and also decides which back end to use depending on the particular platform.

```
#include "dld.h"

#include "base.h"

Base* base_ptr; // 'base_ptr' can point to an instance of a derived class

int main(int argc, char** argv) {
  /* Create a dynamic linker object
     and initialize it with the name of this executable */

  Dld* dyn_linker = new Dld (argv[0]);

  /* Dynamically link in an object file which contains
     the definition of class 'Derived' */

  int status;
  if ((status = dyn_linker->link(argv[1]))==0) {
    // We have successfully linked the object file

    /* Get the address of function 'create_instance' which creates
       an instance of 'Derived' and makes 'base_ptr' point to
       that instance. */
    typedef void (*FP) ();
    FP func_ptr;
    func_ptr = (FP) dyn_linker->get_function("create_instance__Fv");

    // Invoke the function 'create_instance' through the pointer

    if (func_ptr)
      (*func_ptr)();

    /* Now 'base_ptr' points to a 'Derived' instance
       Invoke virtual function 'describe()' on 'base_ptr' */

    base_ptr->describe();

    // Unlink the object module

    dyn_linker->unlink(argv[1]);
  }
  return 0;
}
```

Figure 2-2: Example illustrating the *Dld++* functionalities

2. Each back end provides a set of callback routines which the front end can call to maintain its canonical data structures.

BFD uses the following abstraction for an object file:

1. a header containing information about the rest of the file
2. several sections containing data such as program code
3. relocation information
4. symbol information

All BFD applications revolve around the basic type `bfd`. When an application such as a linker successfully opens a target file (whether it be an object file or archive or any other kind of object file), a pointer to a BFD object (an object of type `bfd`) is returned. All operations on the target object are applied as methods to the BFD object. The type `bfd` has the several data members which maintain information about the target object. The following is a partial list of the data members:

1. The field 'filename' contains the filename of the target file.
2. The 'format' field describes the type of the target file (object, core etc).
3. The 'sections' field points to a linked list of sections and 'section_count' is the number of sections.
4. In the case of object files, 'start_address' field contains the virtual memory address of starting location of the file.
5. In the case of archive library files, 'archive_head' field points to the head of a linked list of member files.
6. The field 'outsymbols' points to the symbol table.
7. The 'xvec' field points to a structure 'bfd_target' which is the target jump table (i.e. a table of pointers to back end functions which perform the actual low-level operations on the object files). When an application wants to perform a certain operation on a target object, BFD translates the application's request into a call to a back end routine through the transfer vector 'xvec'. In short, the transfer vector is the interface between the frontend and a backend of BFD. BFD provides macros to dispatch application's calls to proper back end functions through the 'xvec' member. For example, the following macro simply sends a 'message' to a bfd with a list of arguments 'arglist'.

```
#define BFD_SEND(bfd, message, arglist) \
                    ((*((bfd)->xvec->message)) arglist)
```

2.2.2 Linker Routines

BFD provides three special entry points in the target vector which are useful in applications such as linkers. The three entry points are listed below:

1. The first entry point `_bfd_link_hash_table_create` creates an instance of a symbol hash table which is used by the other linker routines.

2. The entry point `_bfd_link_add_symbols` adds symbols from a file to the hash table created by the above entry point. In the case of object files, the entry point adds all the external symbols of the object file. The actual work of adding the symbol to the hash table is normally handled by the function `_bfd_generic_link_add_one_symbol`. In the case of archive files, the entry point looks through the symbol table of the archive and decides which members of the archive should be included in the linking process. For each member of the archive that is selected, the entry point adds the symbols from that file to the symbol hash table.

3. The entry point `_bfd_final_link` is responsible for the actual linking of all input files. It relocates the contents of the input sections and copies the data into output sections. It also builds an output symbol table.

2.2.3 A Simple BFD application

In this section, we present a BFD-based application which illustrates several functionalities of the BFD library. The application is a rudimentary dynamic linker which can load an object file containing a single function definition into a process and invoke that function through a pointer. Figure 2-3 shows the definition of a function `load` which takes the name of an object file as an argument and loads that object file into the process.

The `load` function creates a `bfd` object, called `input` for the input file, using the function `bfd_openr`. The function `bfd_openr` takes two arguments, a file name and a target name and opens the file with the specified target and returns a pointer to the created BFD object. The target string describes the particular platform on which this program is to run. In this example, we use a target string `a.out-sunos-big` which specifies that this platform uses the `a.out` object file format on a Sun-3 or Sun-4 machine and uses the big endian byte ordering. The call to the function `bfd_check_format` checks that the input file is actually an object file (that is, the file is of the type `bfd_object`). The function then scans through the linked list of sections attached to the input bfd and computes the sizes of the `.text`, `.data` and `.bss` sections and also the total size of these sections. It then allocates memory of size `total_sz` bytes on the heap. At this point, the functions needs to link the input file against the symbol table of the current executable (file `a.out`). Most of the traditional system linkers, such as the GNU linker *ld* [5], provide an incremental loading option. In the case of GNU ld, it provides a command-line switch '-R name' which specifies that the linker has to take 'name' as the name of a file whose symbol

```
#define MAXLEN 256
#include "bfd.h"
/* name of the target such as a.out-sunos-big etc */
char* target = "a.out-sunos-big";
bfd_vma load (const char* filename) {
  asection* sec_ptr; /* a pointer to a section */
  int text_sz, /* size of text section */
      data_sz, /* size of data section */
      bss_sz,  /* size of bss section */
      total_sz; /* total size of all the sections in the object file */
  bfd_vma load_addr; /* address at which file is loaded */
  char buf[MAXLEN];
  bfd* input,*output;
  /* open a bfd for the input file */
  input= bfd_openr(filename,target);
  bfd_check_format(input,bfd_object);
  /* compute the total size of text, data and bss sections */
  for (sec_ptr = input->sections; sec_ptr != NULL; sec_ptr = sec_ptr->next)
    if (strcmp(sec_ptr->name,".text")==0)
      total_sz += (text_sz = sec_ptr->_raw_size);
    else if (strcmp(sec_ptr->name,".data")==0)
      total_sz += (data_sz = sec_ptr->_raw_size);
    else if (strcmp(sec_ptr->name,".bss")==0)
      total_sz += (bss_sz = sec_ptr->_raw_size);
  load_addr = malloc(total_sz);/* allocate storage for the sections on the heap */
  /* invoke the system linker gld */
  sprintf(buf,
    "gld -N -Ttext %X -Tdata %X -Tbss %X -R a.out %s -o a.out.new",
          load_addr,load_addr+text_sz,load_addr+text_sz+data_sz,filename);
  system(buf);
  /* open a bfd for the file a.out.new */
  output = bfd_openr("a.out.new",target);
  bfd_check_format(output,bfd_object);
  /* load the text, data and bss sections into memory */
  bfd_get_section_contents(output,
                          bfd_get_section_by_name(output,".text"),
                          load_addr,0,text_sz);
  bfd_get_section_contents(output,
                          bfd_get_section_by_name(output,".data"),
                          load_addr+text_sz,0,data_sz);
  bfd_get_section_contents(output,
                          bfd_get_section_by_name(output,".bss"),
                          load_addr+text_sz+data_sz,0,bss_sz);
  return load_addr;/* return the starting address of the text section */
}
```

Figure 2-3: A Simple BFD-based Application

table has to be taken as the basis for incremental linking. The file itself should not be included or relocated in the output. This allows the output file to refer symbolically to absolute locations of memory defined in other programs. In our example, the base file is `a.out` whose symbols are taken as the basis for incremental linking and the output file is `a.out.new`. It also uses the command line switches `-Ttext` *text_start_addr* etc to specify where the text, data and bss sections should start in the output file. The system command creates the output file `a.out.new` which contains the sections from the input file, but contains symbols from both the `a.out` file and the input file. The program now creates a bfd object for the output file `a.out.new` and reads the contents of the three sections into memory at appropriate locations by invoking the function `bfd_get_section_contents`. Finally, it returns the start address of the text section. It is assumed that the function definition starts at the beginning of the input file.

The following main program invokes the `load` function with the argument `fact.o` which is the compiled definition of a factorial function.

```
int main() {
  int n=5,fact;
  typedef int (*FP) (int);
  FP func_ptr;
  bfd_init();
  func_ptr = (FP) load ("fact.o");
  fact = (*func_ptr)(n);
  printf("factorial of %d = %d\n",n,fact);
  return 0;
}
```

The `load` function loads the file into memory and returns the entry address of the factorial function into the function pointer `func_ptr`. The main program can now invoke the factorial function through this pointer.

2.3 Implementation issues

In this section we discuss about the implementation of *Dld++*. The design of *Dld++* borrows lots of ideas from the GNU likner *ld* [5] whose source code is available freely. There are two basic classes:

1. Class `file_entry` (Figure 2-4) encapsulates a BFD object. There are two subclasses of this class, namely, class `object_file` and class `archive_file`. The `archive_file` class maintains a list of `file_entry` objects corresponding to its member files.

2. Class `Dld` is the dynamic linker class.

The class `Dld` (Figure 2-5) maintains a list of `file_entry` objects, `input_files` which participate in the dynamic linking. The member `hash` points to the linker's symbol hash table. The member `callbacks` points to a structure of pointers to callback functions. This structure is passed to the BFD routines, and a BFD rountine

```
#include "bfd.h"
#include "List.h"

// generic file_entry class
class file_entry {
  const char* filename; // Name of this file
  /* Name to use for the symbol giving address of text start */
  const char *local_sym_name;
  bfd* the_bfd; // pointer to the BFD object
  /* Symbol table of the file. */
  asymbol **asymbols;
  unsigned int symbol_count;
public:
  file_entry(const char*);
  virtual ~file_entry();
};

// entry for an object file
class object_file: public file_entry {
  boolean just_syms_flag;
  /* reference count - number of entries referenceing this file */
  int ref_count;
  /* Start of this file's text seg in the core */
  bfd_vma text_start_address;
  /* Start of this file's data seg in the core */
  bfd_vma data_start_address;
  /* Start of this file's bss seg in the core */
  bfd_vma bss_start_address;
  /* 1 if this module has all external references resolved */
  boolean all_symbols_resolved_flag;
public:
  object_file(const char* name,int syms_only);
  virtual ~object_file();
  boolean executable_p() {
    return all_symbols_resolved_flag;
  }
};

// entry for an archive file
class archive_file: public file_entry {
  List<object_file> subfiles;
public:
  archive_file(const char* name);
  virtual ~archive_file();
};
```

Figure 2-4: The `file_entry` class definition

invokes these callback functions whenever it needs to inform the linker about a certain event. For example, if the BFD routine which adds a symbol to the linker hash table detects a multiple definition for a symbol, it invokes the call back function `multiple_definition` with appropriate arguments. We will describe more about linker callbacks in the later sections.

2.3.1 Member functions of Class `Dld`

Figure 2-2 already showed several member functions of the class `Dld`. Here, we describe the implementation of these methods. Figure 2-6 shows the definitions for some of the member functions.

1.Constructor The constructor (Figure 2-6) for class `Dld` starts with a call to the function `bfd_init ()` to initialize the BFD libraries. It then creates a `file_entry` object for the input file on its collection `input_files`. It then creates a new instance of a symbol hash table by invoking the function `bfd_link_hash_table_create` and loads the symbols from the input file into the symbol table.

2.link The `link` member function creates a `file_entry` instance for the input file and loads the symbols from that file into the linker's symbol table. It then invokes the BFD entry point `bfd_dlink` which performs the actual linking of the input file. This entry point loads the sections from the input file into memory and relocates their contents by invoking the BFD function `_bfd_relocate_contents_`. Finally, it will scan through all the input files and performs any further necessary relocations. For example, if linking a module resolves a symbol in a module that has already been linked, we need to perform relocation on that module.

3.unlink The `unlink` method checks if the reference count for the input file has reached zero. If so, it will invoke the function `bfd_ulink` which performs the actual unlinking. This function will remove those symbols from the symbol table which are defined by the input file. Then it will scan through the input files and undo the relocations in those files which are affected by the unlinking.

4.get_function This method looks up the symbol table and returns the value of the function symbol requested.

5.executable_p This method checks if all the external symbols are resolved in the particular file. If so, it returns TRUE. This means, all the functions defined in that file can safely be executed.

6.create_reference This method explicitly creates a reference to the given symbol in the linker's hash table. This function is useful if we want to forcefully link a member module from an archive library file.

```
#include "file_entry.h"


// a structure holding a set of callbacks to linker functions
struct bfd_link_callbacks;

// class for the dynamic linker Dld++

class Dld {
  // list of input files involved in the link
  List<file_entry> input_files;

  // Hash table handled by BFD.
  struct bfd_link_hash_table *hash;

  /* Function callbacks. */
  const struct bfd_link_callbacks *callbacks;

public:
  Dld (const char* filename);
  virtual ~Dld ();
  // dynmically link in a file
  int link (const char* filename);
  // unlink a file
  int unlink (const char* filename);
  // get the virtual memory address of a function symbol
  bfd_vma get_function (const char* funcname);
  // get the virtual memory address of a symbol
  bfd_vma get_symbol (const char* funcname);
  // determine if a module has been completely linked
  boolean executable_p(char* filename);
  int undefined_sym_count;
  // return a list of undefined symbols from the symbol table
  char** list_undefined_sym();
  // explicitly create a reference in the symbol table for the given symbol
  int create_reference (char* name);
  // explicitly define a symbol in the symbol table
  int define_symbol (char* name);
};
```

Figure 2-5: The Dld class definition

```
#include "Dld.h"

// The constructor
Dld::Dld(const char* filename) {
  // initialize the BFD library
  bfd_init ();
  // Initialize the linker callbacks
  callbacks = initialize_callbacks();
  // make a file_entry object for the input file
  object_file* my_entry = new (input_files) object_file(filename);
  // Open a BFD for the output file
  output_bfd = open_output_bfd();
  // create a new symbol hash table
  hash = bfd_link_hash_table_create (output);
  // load the symbols from filename into symbol hash table
  bfd_link_add_symbols (my_entry->the_bfd);
}

// The dynamic link function

int Dld::link (const char* filename) {
  // make a file_entry object for filename
  object_file* my_entry = new (input_files) object_file(filename);
  // load the symbols from filename into symbol hash table
  bfd_link_add_symbols (my_entry->the_bfd);
  // call the special BFD entry point for dynamic linking
  bfd_dlink(input_files,output_bfd);
  return 0;
}

// The dynamic unlink function

int Dld::unlink (const char* filename) {
  objfile* file = get_file_entry (filename);
  if (file->ref_count==0)
    delete file;
  bfd_ulink(input_files,output_bfd);
  return 0;
}
```

Figure 2-6: The `Dld` class member functions

2.3.2 Linker Callback functions

The BFD routines need to interact with the linker when a specified event occurs. In such a case, the BFD routine will invoke the appropriate callback function of the linker. The linker callback will take appropriate action for the event and return to the BFD routine. For example, the BFD linker function `_bfd_generic_link_add_one_symbol` adds a symbol to the hash table. When this function detects a multiple definition for a symbol it will invoke the `multiple_definition` callback function of the linker as shown in Figure 2-7.

The linker callback function `multiple_definition` will print a error diagnostic as also the name of the module which first defined that symbol. In the next chapter, we will use this callback mechanism to handle special symbols. For the example, if the BFD routine encounters a special constructor symbol which corresponds to a static object, it will notify the linker through the `constructor` callback function . The `constructor` callback function will place such symbols on a linked list. The linker can later traverse the list and invoke the static constructors at dynamic link time.

2.4 Limitations of the basic *Dld++*

In this chapter, we have presented the design and implementation of a basic version of *Dld++*. It has just the functionalities of *Dld* except that it is portable across object file formats. It cannot yet handle the static scope C++ constructors and destructors. Also if a main program is linked with shared libraries (on systems which support them), then the symbol table of that executable does not have the symbol information for those symbols which are defined in the shared library. For example, if a program is linked with the standard C shared library `libc.so`, then the resulting executable's symbol table does not have entries for function symbols such as `printf` even though these symbols are defined in the shared library. This is because, the shared library itself is dynamically linked to the executable at program start up time by invoking the system's run-time link editor (such as `ld.do` on SUN platforms). We need to extend *Dld++* to handle such issues. The reason why these are not handled in the basic *Dld++* itself is that these issues are either compiler-specific or specific to a particular operating system. In the next chapter, we will show how *Dld++* can be extended to handle these issues depending on a particular C++ compiler or a particular implemantation of shared libraries.

```
boolean
_bfd_generic_link_add_one_symbol (info, abfd, name, flags, section, value,
                                  string, copy, collect, hashp)
{
  /* ... */
  switch (action) {
  case MDEF:
    /* Handle a multiple definition */
    if (! ((*info->callbacks->multiple_common)
               (info, name,
                h->u.c.section->owner, bfd_link_hash_common, h->u.c.size,
                abfd, bfd_link_hash_defined, (bfd_vma) 0)))
          return false;
    /* other cases */
  }
}

/* This is called when BFD has discovered a symbol which is defined
   multiple times. */

/*ARGSUSED*/
static boolean
multiple_definition (info, name, obfd, osec, oval, nbfd, nsec, nval)
     struct bfd_link_info *info;
     const char *name;
     bfd *obfd;
     asection *osec;
     bfd_vma oval;
     bfd *nbfd;
     asection *nsec;
     bfd_vma nval;
{
  einfo ("%X%C: multiple definition of '%T'\n",
         nbfd, nsec, nval, name);
  if (obfd != (bfd *) NULL)
    einfo ("%D: first defined here\n", obfd, osec, oval);
  return true;
}
```

Figure 2-7: A Linker Callback Function

Chapter 3

Basic Extensions to *Dld++*

In this chapter, we show how *Dld++* can be extended by subclassing from the `Dld` class. To demonstrate this, we consider two issues. The first issue concerns itself with handling the constructors and destructors of static scope C++ objects in an object module. The second issue is about handling the shared libraries.

3.1 Dynamic Initialization of Static Objects

When an object module is dynamically linked into a program, constructors should be invoked on the static scope objects defined in that module. Similarly, when the module is unlinked, the corresponding destructors should be invoked on those objects.

Before we explain how this is handled in *Dld++*, we briefly describe how this problem is solved in the case of statically linked C++ programs. Each C++ compiler handles this problem in its own way. We take the case of a particular C++ compiler called g++ [17]. The compiler generates special symbols for these initialization functions. The initialization functions will have a prefix `__GLOBAL_$I$` and the termination functions will have the prefix `__GLOBAL_$D$`. The linker must build two lists of these functions–a list of initialization functions, called `__CTOR_LIST__`, and a list of termination functions, called `__DTOR_LIST__`. The GNU linker *ld* builds these lists in the following way. The linker provides a callback function `constructor_callback` to the BFD routines. The BFD function `_bfd_generic_link_add_one_symbol` adds a given symbol to the linker's hash table. When this routine discovers a special constructor symbol, it will invoke the linker callback. The callback function accumulates such symbols in the two lists: `__CTOR_LIST__` and `__DTOR_LIST__`.

Depending on the operating system and its executable file format, either 'crtstuff.c' or 'libgcc2.c' traverses these lists at startup time and exit time. Constructors are called in forward order of the list; destructors in reverse order.

We follow a similar approach in the case of *Dld++*. We define the function `is_special_symbol()` which takes a symbol name as argument and returns a non-zero value if the symbol is a special symbol; else it returns zero. A pointer to this function is passed to the `_bfd_generic_link_add_one_symbol` function along with other callbacks. This BFD routine uses the function to decide if a symbol is a special symbol. If so, it will call the constructor_callback function of the linker.

We define a subclass gcc_dld of class Dld as shown in Figure 3-1. The class

```
enum special_sym_type {CTOR=1,DTOR} ;

special_sym_type
is_special_symbol(char* s) {
  if (has_prefix(s,"__GLOBAL_$I$"))
    return CTOR;
  else if (has_prefix(s,"__GLOBAL_$D$"))
    return DTOR;
  else
    return 0;
};


// File gcc_dld.h for class gcc_dld
#include "Dld.h"

/* Declare a pointer to void function type. */
typedef void (*func_ptr) (void);

class gcc_dld : public Dld {
// list of constructors
  func_ptr* __CTOR_LIST__;
public:
  add_special_symbol(special_sym_type,bfd_vma,file_entry*);
  reset_ctor_list();
  int gcc_link(char*);
  int gcc_unlink(char*);
};

class gcc_file: public object_file {
 // maintain a list of destructor symbols
  func_ptr* __DTOR_LIST__;
};
```

Figure 3-1: Class gcc_dld

gcc_dld maintains a list of function pointers __CTOR_LIST__ which correspond to the constructor symbols found during a call to its method gcc_link. The callback constructor_callback invokes the method add_special_symbol. If the symbol is a constructor symbol, it is placed on the __CTOR_LIST__. If it is a destructor symbol, it is placed on the list __DTOR_LIST__ of the corresponding gcc_file object.

Figure 3-2 shows the methods gcc_link and gcc_unlink. The method gcc_link invokes the base class method Dld::link. It then traverses the __CTOR_LIST__ and

```
// file gcc_dld.c
#include "gcc_dld.h"

// method gcc_link

int gcc_dld::gcc_link(char* name) {
  // invoke the parent class link method
  Dld::link(name);
  // traverse the list of constructors and invoke them
  for (int i=0; __CTOR_LIST__[i] !=0; i++)
    __CTOR_LIST__[i] ();
  // reset the ctor list
  reset_ctor_list();
  return 0;
}

int gcc_dld::gcc_unlink(char* filename) {
  // get the file_entry
  gcc_file* file = get_file_entry (filename);
  // traverse the list of destructors of the file_entry
  for (int i=0; file -> __DTOR_LIST__[i] !=0; i++)
    file -> __DTOR_LIST__[i] ();
  // call the base class unlink
  Dld::unlink(filename);
  return 0;
}
```

Figure 3-2: Methods of class gcc_dld

invokes those initialization functions. After each call to gcc_link, the __CTOR_LIST__ is reset. The method gcc_unlink obtains the gcc_file object for the given file. It then traverses the __DTOR_LIST__ of that object and invokes those termination functions. It finally calls the base class method Dld::unlink.

We now present an example program which demonstrates the application of class gcc_dld in the implementation of an idiom of programming known as the *Virtual Constructor* idiom.

3.1.1 The Virtual Constructor Idiom

In C++, one cannot declare a constructor to be virtual [8]. It is assumed that, everything that is needed to create an object is known to the programmer at the point where an object is created and there is no object until after it has been created. The virtual function mechanism is provided to allow operations to be invoked on those objects whose exact type is unknown to the programmer.

The assumption (that everything that is needed to create an object is known at the point of its creation) is not valid in a scenario where new classes are added to a program at run time. In such a case, the program would like to create an object of a class that it does not know about. All the program knows about the type of the object is that its type is a sub type of a known type. There are several techniques to solve this problem. These techniques are generally known as *virtual construction* techniques.

In this section, we present a small example which demonstrates this technique (Figure 3-3). The idea is borrowed from the *autonomous generic exemplar idiom* discussed in [6].

The program shows a root class which has a virtual method clone () which can create new instances of a class on the heap. The main program maintains exemplar_list which is a collection of instances whose types are derived from root. The file derived.c defines a subclass derived of class root and defines the virtual method clone. This method creates an instance of derived on the heap and returns a root* pointer to the created instance. The file also defines an exemplar instance of derived on the exemplar_list. This exemplar is defined in the global scope. When this file is compiled by g++, the compiler would generate special constructor and destructor symbols for such static objects. In this example, the compiler generates the symbol __GLOBAL_$I$exem for static constructors and the symbol __GLOBAL_$D$exem for static destructors.

In the main program, shown in Figure 3-4 , we make use of the class gcc_dld. The extended dynamic linker links in the object file derived.o by using the method gcc_link. This would automatically create an exemplar instance for the class derived on the global collection exemplar_list. The program can find the exemplar instance for a particular derived class by invoking the function find_exemplar with the name of the class as its argument. Once we find the exemplar for a class, we can create new instances of that class by invoking the virtual method clone on that exemplar. Thus, we have shown how a program might create instances of an unknown class in a type-safe way.

```
// File root.h
#include "String.h"
class root {
  String name;
public:
  root (char* n) {name = n;} // constructor
  ~root(){} // destructor
  virtual root* clone (int n)==0; // virtual clone method
  virtual void f(); // a generic virtual function
};

#include <List.h>

// maintain a list of exemplar instances
List <root> exemplar_list;

// File derived.c – a derived class definition
#include "root.h"

class derived: public root {
  int y;
public:
  derived(char* n):root(n){} // constructor
  derived(int i){y=i;} // another constructor
  ~derived() {} // destructor
  virtual root* clone (int n) {
    // create a 'derived' instance on heap
    return new derived (n);
  }
  virtual void f(){}; // a generic virtual function
};

/* create a global exemplar instance of 'derived'
   on the 'exemplar_list' */

derived exem ("derived");
root* temp = exemplar_list.insert(&exem);
```

Figure 3-3: The Virtual Constructor Idiom

```
// File main.c – the main program
#include "gcc_dld.h"
#include "root.h"

int main(int argc, char** argv) {
  gcc_dld* dyn_linker = new gcc_dld (argv[0]);
  // link in the object file 'derived.o'
  int status;
  if ((status = dyn_linker->gcc_link(argv[1]))==0) {
    /* find the exemplar instance by name for class 'derived'
       in the 'exemplar_list' */
    root* exemplar = find_exemplar("derived");
    if (exemplar) {
      // create a new instance of 'derived'
      root* inst = exemplar->clone(1);
      // 'inst' now points to a 'derived ' instance
      inst->f(); // invokes derived::f()
    }
    // unlink the module derived.o
    dyn_linker->gcc_unlink(argv[1]);
    // remove the exemplar instance
    remove_exemplar("derived");
  }
  return 0;
}
```

Figure 3-4: The Virtual Constructor Idiom - Main program

3.2 Shared Libraries

The basic version of *Dld++* cannot handle shared libraries. This means that if the main executable was dynamically linked (that is, it was linked against some shared libraries), its symbol table will not contain symbols exported by those shared libraries. For example, if the main executable was linked with the shared library version of the standard C library (libc.so on Sun4 platforms), its symbol table will not have entries for such common functions as `printf`. This is because, the shared library, libc.so which defined this function was dynamically linked to the main program at its startup time. Suppose we try to dynamically link a module using `Dld::link()` method. If the module references the `printf` function, then the `Dld::link()` method will report an undefined symbol error for that function. One way out of this is to make the main executable statically linked so that its symbol table is complete. But this is not a feasible solution in the case of large executables. Hence, we need to extend the functionalities of the dynamic linker to handle the shared libraries also.

In this section, we will show *Dld++* can be extended to handle the shared libraries in SunOS [9]. We briefly describe how shared libraries are implemented in SunOS before presenting the solution.

In the address space of a dynamically linked executable, the linker *ld* creates an instance of a `link_dynamic` structure with the symbol name `__DYNAMIC`. The `link_dynamic` structure is used by the execution-time link editor *ld.so* to obtain all the needed shared objects.

The program bootstrap routine `crt0` checks if the symbol `__DYNAMIC` is defined; if so the executable needs further link editing and so it transfers control to *ld.so*. *ld.so* processes the information contained in the `__DYNAMIC` structure of the program in order to complete the link editing required to start the program. In particular, *ld.so* obtains the list of shared objects that must be added to the address space of the process and link edited. *ld.so* looks up the shared object and maps it into the process's address space. For each shared object that is mapped, it builds a `link_map` structure of the following structure:

```
/ * Structure describing name and placement of dynamically loaded
  * objects in a process' address space. */
struct link_map {
caddr_t lm_addr; /* address at which object mapped */
char  *lm_name; /* full name of loaded object */
struct link_map *lm_next; /* next object in map */
struct link_object *lm_lop; /* link object that got us here */
caddr_t lm_lob; /* base address for said link object */
int lm_rwt : 1; /* text is read/write */
struct link_dynamic *lm_ld; /* dynamic structure */
caddr_t lm_lpd; /* loader private data */
};
```

Each such structure is placed on a singly linked list (linked through the `lm_next` field) and the head of the list is rooted in the `ld_needed` field of the program's `__DYNAMIC` structure. At this point, *ld.so* attempts to complete the link editing that was begun by *ld* at link time.

In order to handle the shared libraries during the dynamic linking, we only need to modify the constructor of class `Dld`. In our implementation, We reuse some functions from the GNU debugger GDB [18]. The basic steps performed by the modified `Dld` constructor are described below.

1. Locate the base address of the SunOS dynamic linker structures, i.e., locate the symbol `__DYNAMIC`. This is very trivial in SunOS, because this symbol is already in the symbol table of the executable. We just need to look it up there.

2. Locate the list of shared objects that are mapped to this process's address space. We just need to identify the head of the linked list of `link_map` structures. We can use the GDB routine `find_solib` to step through the list of shared objects that are loaded into the current process.

3. We now have information about where each shared object is mapped in the address space. We have to enter the symbols from each shared object into the linker's symbol hash table. We can write a routine similar to the GDB routine `solib_add` which adds symbols from a shared library file to the global symbol table.

After having done this, the dynamic linker's symbol table contains symbol information from the shared objects also. From now onwards, the methods such as `Dld::link` can make use of this symbol information.

It is also fairly straightforward to map new shared objects to the process's address space.

Chapter 4

A Dynamic Linker for E

In this chapter, we show how to build a dynamic linker for the persistent C++ language **E**. This necessitates a few extensions not only to the basic *Dld++* presented in Chapter 2, but also to the run-time support library of the **E** language itself.

E [13] is an extension of C++ language providing database types and persistence. Persistence in **E** entails some form of dynamic linking of method code. This is because a program might encounter, on the persistent store, an object whose type was not known to the program when it was compiled. This necessitates dynamic linking of the method code of the corresponding type to the program so that the type definition is made available to the program. The current run-time support library provided by **E** is inadequate for this purpose. We modify and extend the run-time library of **E** by adding functionalities to dynamically link and unlink object modules. We then present the design of a class `Type` to facilitate *persistent types.* Each user-defined type will have a unique persistent `Type` object associated with it. Class `Type` provides methods for dynamic linking and unlinking of user-defined classes using the extended run-time support. In addition, class `Type` ensures identity of user-defined types, i.e., each user-defined type will have a unique identity across compilations of a program.

4.1 Overview

We have seen in Chapter 1 that incremental linking is desirable in certain applications and it is possible to incorporate such a facility into an object oriented system given the availability of a dynamic linker such as *Dld++*. As mentioned in that chapter, the basic problem with incremental linking in C++ is to provide a technique for adding a new class to a running program. To add a new class to a running program, we need to address the following issues:

1. We should be able to create instances of the class, which is yet to be defined, from the original program.

2. We should be able to invoke operations on objects of the new class from the original program in a type-safe manner.

But when we add persistence to C++, we have to address a new problem . A program should be able to invoke operations on objects that are *already* existing on a persistent store. It may so happen that the program does not know the type of an object that it encounters on a persistent store because that instance was created by another program which shares the persistent store with the original program. Hence in the case of persistent languages, some form of dynamic loading of method code is necessary to handle such situations.

E is a version of C++ designed for writing software systems to support persistent applications. **E** augments C++ with database types, iterators, collections and transaction support. **E** mirrors the existing C++ types and type constructors with corresponding database types (db-types) and db-type constructors. Db-types are used to describe the types of objects in a database. In particular, a *dbclass* is the database counterpart of the normal C++ class. Only the instances of a dbclass are allowed to be persistent. **E** provides a new storage class **persistent** which is the basis for creating objects on a persistent store. If the declaration of a named db-type object specifies that its storage class is **persistent**, then that object retains its value between executions of the program and it survives any crashes of the program. For transaction support, **E** provides library calls to begin, commit or abort a transaction. These calls are supported by the EXODUS storage manager [3]. A persistent object can be shared among several programs. This is achieved by linking the object module containing the definition of the persistent object to any program which needs to have access to that persistent object.

The current implementation of **E** has some shortcomings as discussed in [13]. These problems basically arise from the standard C/UNIX model of compiling and linking programs which is insufficient to support a persistent language like **E**. We describe these problems briefly below:

1. **Type Identity.** **E** does not maintain persistent types. Programs share classes by textual inclusion of the corresponding header files. Suppose a program creates a persistent object of type `T` and later another program with a different definition for `T` manipulates this object. In such a case, the database can easily be corrupted. That means **E** should maintain the identity for each user-defined type. In addition, each persistent object should carry sufficient information to identify its type. Hence object persistence entails *type persistence.* The current implementation of **E** provides only an approximation to type identity. For each user-defined class, the compiler computes a hash value from the class definition. This hash value serves as a type tag and is stored in each persistent instance of the class. The type tag serves in identifying the appropriate virtual function table at run-time when a virtual function is invoked on the object.

2. **Type Availability.** Often it is possible for a program to encounter an object on the persistent store whose type was not known at the time the program was compiled. Suppose a program **P1** creates a persistent collection `p_c` whose elements are of type `B`. **P1** traverses `p_c` and invokes a virtual function `f()` on each element of `p_c`. Another program **P2** defines a new type `D` which is

derived from B and then adds an instance of type D to p_c. If **P1** is run again, an "unknown type" error occurs when it tries to invoke the virtual function f() on the newly added instance. Neither the virtual table nor the method code for the class D is present in the address space of **P1**. So, whenever an instance of a new sub-type is added to p_c, the program **P1** needs to be updated by linking the code for the new type to **P1**. This is not desirable for two reasons: (a) it is not in keeping with good object-oriented design; and (b) programs **P1** and **P2** might be running concurrently. Hence, we conclude that dynamic linking of method code is necessary in such situations.

In this chapter, we address these issues and present solutions for the problems of type identity, type persistence and type availability in **E**.

4.2 Run-time Environment in E

In this section, we briefly describe the run-time support provided by **E** to handle virtual function dispatch and initialization of persistent objects. (The implementation of persistence in **E** is discussed in detail in [12]). In the next section we extend this run-time support to provide a mechanism for dynamic linking and unlinking of object modules. We also modify the virtual function dispatch mechanism to facilitate dynamic linking of classes.

4.2.1 Virtual Function Dispatch

C++ implements virtual functions by generating a virtual table (vtbl) for each class having a virtual function. The vtbl is a table of addresses of appropriate virtual methods. Every instance of such a class will have an embedded pointer (vptr) to the vtbl of its class. At run-time, invocation of a virtual function on an instance involves an indirection through the vtbl. In the case of persistent instances, the above implementation will no longer suffice, because we cannot store the address of a dispatch table in a persistent instance. That address is valid only for one program and it is not guaranteed that the table will be loaded starting at the same address of the main memory in another program. Several solutions have been proposed to handle virtual function invocation on persistent instances. An approach to this problem has been presented in the context of a persistent object-oriented language O++ in [1]. The solution involves modifying each user-specified constructor. Whenever a persistent instance is read into memory, the vptr is set to the correct vtbl address by invoking the modified constructor on that instance. The modified constructor only 'fixes' the vptr but otherwise leaves the instance unchanged.

E implements a different solution for this problem. For every dbclass having virtual functions, the compiler generates a unique integer type tag and every instance of the dbclass will contain this tag. This type tag is generated by computing a hash value from the class definition; same class definition hashes to the same tag in different compilations. **E** introduces a global hash table for mapping the type tags to virtual

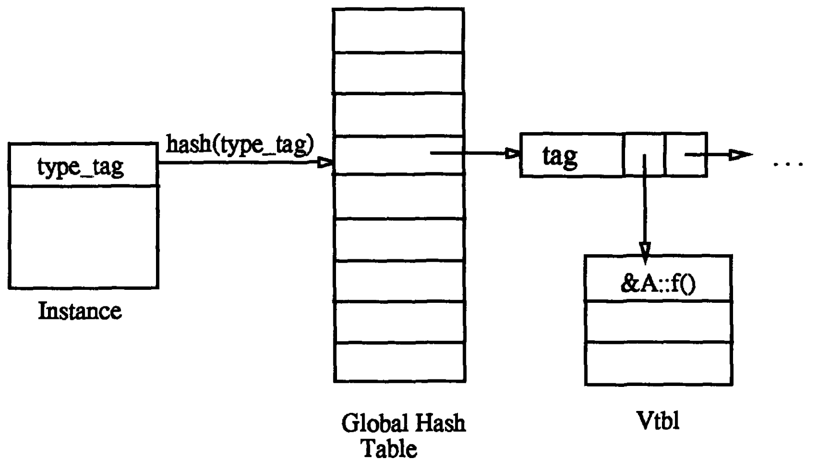

Figure 4-1: Virtual Function Dispatch in **E**

table addresses. To call a virtual function, the hash function is applied on the type tag to get the vtbl address and then proceed as before by indexing into the virtual table. Figure 4-1 illustrates the virtual function mechanism of **E**. The run-time support library of **E** provides a function `dbGetVtbl` which returns the address of the virtual table (vptr) given the address of a db-instance (Figure 4-2). `dbGetVtbl` hashes on the integer type tag (stored in the instance) to find the hash table index. If the correct entry is found at that index, it extracts the vptr and returns it. Otherwise, it aborts the currently running transaction. The global hash table is initialized at

```
void *  dbGetVtbl( this_pt, vpt_offset)
pDBREF this_pt;
int vpt_offset; /* offset of vptr field from this_pt */
{
/*
1. get type tag value by dereferencting this_pt.
2. hash into the E_hash table to get
   the entry for the correct virtual table.
3. if an entry is found then
       return the pointer to virtual table (vptr)
   else
       call _E_abort to abort the transaction
*/
}
```

Figure 4-2: The `dbGetVtbl` Function

program start-up to make entries for all the dbclasses in a particular program. The mechanism for initialization and destruction of static objects is extended to also initialize the hash table. This is described in Section 4.2.3.

4.2.2 Initialization of Persistent Objects

In C++, the initialization of non-local static objects in a translation unit is done before the first use of any function or object defined in that translation unit [8]. It is generally done before the first statement of `main()`. Destructors for initialized static objects are called when returning from `main()`. These actions are performed in each execution of a program. In **E**, persistent objects have to be handled differently from transient objects with respect to initialization and destruction. A persistent object has to be initialized only once in its life time and it must be initialized before any program actually uses it. In general, all the persistent objects declared in an E source module are initialized when the corresponding object module is linked to any program for the first time. For each source module, **E** maintains a persistent flag indicating whether the persistent objects in that module have already been initialized or not. The run-time support provides a function `E_Pinit` to perform this one-time initialization of persistent variables.

4.2.3 Start and Termination of an E Program

The current version of **E** extends GNU C++ [17] to provide support for persistence. In this section, we explain briefly how **E** extends the static initialization mechanism of GNU C++ for initializing persistent variables and the global hash table (used in virtual function dispatch). GNU C++ employs different techniques (depending on the operating system and executable file format) to handle the initialization and destruction of static variables. One of the techniques, which can easily be extended to dynamic linking, uses a program called `collect2` during the linkage step of the object modules. The `collect2` program operates on a set of object files to collect static initialization and destruction information from them. It builds two lists: a list of initialization functions, `__CTOR_LIST__`, and a list of termination functions, `__DTOR_LIST__` These lists are generated as C code and the C code is compiled and linked along with the rest of the object modules. At program start-up, the `__CTOR_LIST__` is traversed in the forward order to invoke all the static constructors; at program exit time, the `__DTOR_LIST__` is traversed in the reverse order to invoke the global destructors.

E extends the `collect2` program to also collect information about static persistent variables and virtual tables and the information is stored in the lists `__PTOR_LIST__` and `__VTOR_LIST__` respectively.

At the start-up time of an **E** program, the **E** run-time library function `E_main()` performs the static initialization through the following sequence of actions:

1. Invoke all the static constructors by scanning the list of pointers to constructors in the list `__CTOR_LIST__`.

2. Initialize all the static persistent variables by scanning the list __PTOR_LIST__ (this is handled by the **E** run-time function `E_Pinit` mentioned earlier).

3. Scan the list __VTOR_LIST__. For each entry found in the list, find the corresponding integer type tag and make an entry in the global hash table for the type tag.

At the exit time of an **E** program, the **E** run-time library function `_E_exit()` invokes the destructors for the static variables by traversing the list __DTOR_LIST__.

4.3 Extensions to *Dld++*

In order to build a dynamic linker for **E** we define a new subclass `E_dld` of class `Dld`. This subclass is very similar to the subclass `gcc_dld` defined in the previous chapter. In addition to the __CTOR_LIST__, it defines two more lists. The list __VTOR_LIST__ contains the virtual table information and the list __PTOR_LIST__ contains the symbols for the static persistent variables. We also modify the function `is_special_symbol()` defined earlier. The modified function now identifies four types of symbols.

```
enum special_sym_type {CTOR=1,DTOR,VTOR,PTOR} ;

special_sym_type
is_special_symbol(char* s) {
  if (has_prefix(s,"__GLOBAL_$I$"))
    return CTOR;
  else if (has_prefix(s,"__GLOBAL_$D$"))
    return DTOR;
  else if (has_prefix(s,"__GLOBAL_$V$"))
    return VTOR;
  else if (has_prefix(s,"__GLOBAL_$P$"))
    return PTOR;
  else
    return 0;
}
```

The class `E_dld` defines two additional methods. The methods are `E_link` and `E_unlink`. The method `E_link` (Figure 4-3) dynamically links in an **E** module and performs the appropriate initializations. The sequence of actions performed by `E_link` are described below:

1. Dynamic Linking. Invoke the base class method `Dld::link`.

2. Dynamic Initialization. This involves the following steps:

   1. Update the global hash table to make entries for the new vtbls by traversing the list __VTOR_LIST__.

2. Invoke the function `E_Pinit` on the list `__PTOR_LIST__` to initialize the static persistent objects. `E_Pinit` will intialize the persistent objects only if this module is being linked for the first time.
3. Traverse the list `__CTOR_LIST__` to invoke static constructors.

`E_unlink` (Figure 4-4) gets the `gcc_file` object corresponding to the given file name and performs the following:

1.Dynamic Destruction. The sequence of actions performed in this subtask are:

1. Traverse the list `__DTOR_LIST__` of the file entry to invoke the destructors of the static variables.
2. Traverse the list `__VTOR_LIST__` of the file entry to update the global hash table by deleting the corresponding hash table entries.

2. Dynamic Unlinking. Invoke the base class method `Dld::unlink` to unlink the object module from the running process.

We note that in the above sequence it is crucial that the destructors are called before the vtbl is removed from the memory so as to be able to handle virtual destructors.

4.4 Dynamic Linking and Unlinking of Classes

In the previous section, we have described a technique for dynamic linking/unlinking of an **E** object module. But these operations by themselves are not sufficient in an object-oriented language like **E**. This is because of dependencies between classes. For example a dbclass `A` might be a subclass of a dbclass `B`. Hence, before the object code for `A` is loaded, we have to make sure that the object code for `B` has already been loaded into the memory. This is for ensuring the proper construction of an instance of `A` as well as virtual function invocation. Thus we notice that , for each user-defined dbclass, there should be some minimal run-time type information associated with it. This run-time type information can then be used by the load routines to properly load all the required object modules. For dynamic linking purposes, the only type information needed is the list of base class names. The run-time type support is provided by means of defining a class `Type`. For each user-defined dbclass, an persistent instance of `Type` is created which contains the type information about the user-defined class such as the list of names of base classes.

4.4.1 Class Type

The declaration and definition of class `Type` is shown in Figure 4-5. The dbclass `Type` provides methods `load` and `unload` to perform dynamic loading and unloading of a dbclass respectively. The `load` method checks that all the base classes are loaded and then invokes `E_link` to dynamically link the object module to the running process. The `unload` method simply unlinks the object module by invoking `E_unlink`.

```
#include "E_dld.h"

// adds a vtbl entry in the global hash table
extern "C" _E_add_vtbl(struct _E_vthash *);

// file header structure
struct f_header {
        char *file_name;                    // name of the file
        int comp_t;                         // time of compilation
        struct Pvar_entry * Pvars;          // pointer to ptab vector
        int ptab_count;                     // size of Pvar array
};
// hash entry for a virtual table
struct _E_vthash {
        int tag;            /* efront generated type tag */
        struct __mptr * vtbl; /* address of efront -generated vtbl */
        struct _E_vthash * next;            /* next entry in hash table */
        char * str; /* class name; used if collision detected */
};
// function to initialize static persistent variables
extern  void _E_Pinit(f_header** Fhead);

// method to dynamically link an E module
int E_dld::E_link(char* filename) {
  Dld::link (filename);
  // add vtbl entries to the global hash table
  struct __E_hash ** vtpt;
  for (int i=0;__VTOR_LIST__[i] !=0;i++){
    struct _E_vthash* vtp = (struct _E_vthash*)(__VTOR_LIST__[i]);
    _E_add_vtbl(vtp);
    set_Class_tag(vtp->str+4,vtp->tag);
  }
  // initialize static persistent variables
  for (int p=0;__PTOR_LIST__[p] !=0 ;p++)
    _E_Pinit(((f_header**)&(__PTOR_LIST__[p])));
  // invoke the static constructors
  for (int c=0;__CTOR_LIST__[c] !=0 ;c++)
    __CTOR_LIST__[c]();
  // reset the ctor list
  reset_ctor_list();
  return 0;
}
```

Figure 4-3: Dynamic Linking of E classes

```
#include "E_dld.h"

// removes a vtbl entry from the global hash table
extern "C" _E_del_vtbl(struct __E_hash *);

// hash entry for a virtual table
struct _E_vthash {
        int tag;              /* efront generated type tag */
        struct __mptr * vtbl; /* address of efront -generated vtbl */
        struct _E_vthash * next;             /* next entry in hash table */
        char * str; /* class name; used if collision detected */
};

// method to dynamically unlink an E module
int E_dld::E_unlink(char* filename) {
  // get the file_entry
  gcc_file* file = get_file_entry (filename);
  // traverse the list of destructors of the file_entry
  for (int i=0; file -> __DTOR_LIST__[i] !=0; i++)
    file -> __DTOR_LIST__[i] ();
  // delete the virtual table entries from the global hash table
  for(i=0; file -> __VTOR_LIST__[i] !=0 ;i++)
      _E_del_vtbl(file -> __VTOR_LIST__[i]);
  // call the base class unlink
  Dld::unlink(filename);
  return 0;
}
```

Figure 4-4: Dynamic Unlinking of E classes

An alternative (simpler) solution would be to archive all the object modules into an archive library file. Since the `E_link` method searches a library file repeatedly to load as many member files as possible, it would automatically load the object modules corresponding to the base classes whenever a derived class file is linked. But this requires that all the related object modules should be archived into a single library file.

All the `Type` instances are allocated on a global persistent collection:

```
persistent collection<Type> type_collection;
```

All the programs share this `type_collection`.

In the next two sub-sections, we will show how the problems mentioned in Section 1, namely, type identity and type availability can be solved using the `Type` class.

4.4.2 Type Identity

In this section, we describe an idiom for ensuring type identity for user-defined classes. The class `Type` maintains a persistent collection of `Type` instances. It also ensures that at any time there exists exactly one persistent `Type` instance for a given user-defined class. This constraint is enforced by the `create_type` method of `Type` class (see Figure 4-5). Since the constructor for dbclass `Type` is protected, the only way to create `Type` instances is through the static method `create_type`. The `create_type` method scans the `type_collection` to check if there is already a `Type` instance in the collection with the same name. If so, it generates an error message. Otherwise, it creates a new `Type` instance and returns a pointer to that instance. We need a mechanism to install a persistent `Type` object on the type collection for each user-defined class. This `Type` object has to be created the first time the object module containing the type definition is linked (statically or dynamically) to any program. We demonstrate this with a simple example. Consider a dbclass `A`. Figure 4-6 shows the class definition and the relevant code. Each user-defined class should have a static pointer to the appropriate `Type` object. This static pointer has the **persistent** storage class. That means, it is initialized only once in its life-time. It is initialized the first time the object module is ever linked to any program either statically or dynamically. Moreover, the pointer is initialized to the return value of the function `Type::create_type` which enforces the type identity. This way we can ensure that user-defined types have unique identity across compilations of programs.

4.4.3 Type Availability

In this section, we propose two solutions to the problem of type availability discussed in Section 1.

Consider a dbclass `A` and a persistent collection of A instances, `coll_A` which is shared between several programs. Note that, for every user-defined class, there exists one unique persistent `Type` instance on the `type_collection` which is shared by all the programs. Before scanning every shared collection, we have to scan the `Type` collection to ensure that all types are loaded into the program. This way, we can make

```
// File Type.h
dbclass Type {
  String name; // name of the class
  StringList base_class_list; // list of base class names
  dbint type_tag;
 protected:
  Type(String& n,StringList& b =0); // protected constructor
 public:
  ~Type();
  static Type* create_type(String&,StringList&);
  static int load_type(int tag_val); // loads a type definition given tag
  int load(); // load this type
  int unload(); // unload this type
};

// File Type.e

// static method to enforce type identity
#include "Type.h"
#include <E/collection.h>

persistent collection<Type> type_collection;

Type* Type::create_type(String& n, StringList& b)
{
  collection_scan<Type>ts(type_collection);
  Type* tptr;
  while (tptr = ts.next())
    if (tptr->name == n)
      Error("Type already exists!");
  return new (type_collection) Type(n,b);
}

int Type::load_type(int tag_val)
{
  collection_scan<Type>ts(type_collection);
  Type* tptr;
  while (tptr = ts.next())
    if (tptr->type_tag == tag_val)
      return tptr->load();
  return 0;
}
```

Figure 4-5: Definition of Class **Type**

```
#include "Type.c"

// File A.h

dbclass class_A{
   dbint i;
 public:
   A(int i);
   ~A();
   // static member pointer to the Type object
   static persistent Type* type_ptr;
};

// File A.e

#include "A.h"

// method definitions for class_A

// ...

//InitializationInitialization for the static member 'type_ptr'

persistent Type* class_A::type_ptr = Type::create_type("class_A");
```

Figure 4-6: Idiom for Type Identity for a Dbclass A

sure that the program does not encounter any object on the shared collection whose type is not known to it. Figure 4-7 shows the listing for this. Another important thing is that while one program is scanning a shared collection, we need to prevent other programs to create any new Type instance on the type_collection. Thus , the scanning of both the type collection and the shared collection have to be done in a single transaction.

```
#include "A.c"
#include "Type.c"
#include <trans.h>
extern collection<A> a_coll;

main()
{
  E_BeginTransaction();
  // Scan the Type collection
  collection_scan<Type>ts(type_coll);
  Type* t;
  while (t=ts.next())
    t->load();
  collection_scan<A> as(a_coll);
  A* a;
  while (a=as.next())
    a->f(); // f() is some virtual function
  E_CommitTransaction();
}
```

Figure 4-7: A Solution to Type Availability Problem

As we can easily notice, the above solution is a very conservative one. If the type_collection is very large, then scanning it whenever a shared collection is to be scanned could be expensive. Hence we propose a solution which entails modifying the virtual function dispatch mechanism (dbGetVtbl function).

Instead of ensuring all type definitions are properly loaded into a program before scanning each shared collection, we would like to handle the situation only when the need arises. That means, when we realize that a hash table entry is not found for a given type tag, we can arrange to load the corresponding type definition. Class Type provides a static method load_type for this purpose. This function takes a type tag as an argument, finds the corresponding Type object in the type_collection, and loads the object module corresponding to that type. Figure 4-8 shows the modified GetVtbl function. This method does not incur any extra cost for normal virtual function calls. It incurs a one-time cost to load a particular object module when an instance of unknown type is encountered on a shared collection.

```
// Modified dbGetVtbl function

#include "Type.c"

void *   dbGetVtbl(pDBREF this_pt, int vpt_offset)
// 'vpt_offset' is the offset of vptr field from this_pt
{
/*
1. get type tag value by dereferencting this_pt.
2. hash into the E_hash table to get
     the entry for the correct virtual table.
3. if an entry is found then
3a. return the pointer to virtual table (vptr)
     else
3b. call Type::load_type(type_tag_value);
3c. hash into the E_hash table to get
          the entry for the correct virtual table
3d. if an entry is found then
               return the pointer to virtual table (vptr)
          else
               call _E_abort to abort the transaction.
*/
}
```

Figure 4-8: Modified Virtual Function Dispatch Mechanism

4.4.4 MetaClass

So far we have demonstrated a way of dynamically linking and unlinking classes in a **E** program. In this section, we present a method for creating instances of a dynamically linked class and maintaining those instances. We define a dbclass `MetaClass` for this purpose. Just as a `Type` instance manages the type information of user-defined class, a `MetaClass` instance manages the persistent instances of a user-defined class. The class definition of `MetaClass` is shown in Figure 4-9.

A persistent `MetaClass` instance is created on the persistent collection `meta_collection` for each user-defined class. A `MetaClass` instance for a user-defined class contains a collection of instances, called the *extent* of the class. All user-defined classes are derived from `root` class. We introduce the notion of *persistent exemplar idiom* which is a natural extension of the exemplar idiom discussed in [6] to the domain of persistence (Figure 4-10). A *persistent exemplar* is a designated persistent instance of a persistent type which is created once for the life-time of that type. This persistent exemplar instance can be shared by any program which also shares the corresponding persistent type. That means, a program can create the instance of a dbclass through the corresponding exemplar instance without textually including the type definition for that dbclass. The extended `dbGetVtbl` function (defined earlier) will take care of the dynamic linking of the appropriate object modules into the program. Each `MetaClass` instance contains a pointer to a persistent exemplar instance of the corresponding user-defined class. Both the `MetaClass` and persistent exemplar instances of a dbclass are created the first time the object module containing the class definition is linked (statically or dynamically) to any program. This persistent exemplar instance can then be shared by any program which also shares the `meta_collection`. The static method `create_instance` of `MetaClass` takes a class name as argument and searches the `meta_collection` for the corresponding `MetaClass` instance. If the `MetaClass` instance is found, a new instance is created on the `extent` of the class by invoking the `make()` operation on the exemplar instance of the `MetaClass` instance.

4.5 Summary

We have discussed some important issues related to incremental linking in a persistent object-oriented language like **E**. We have presented a layered approach to the problem of incrementally adding or modifying **E** classes to a running program. The bottom-most layer is a dynamic linker `Dld++` which can perform certain basic dynamic link-editing functions. The method `E_link` dynamically links in an object file into the program, makes an entry in the global hash table for each virtual table found in the object module, allocates and initializes the static persistent variables (if the module is being linked in for the first time), and initializes the static transient variables. The method `E_unlink` destroys the static transient variables, removes the relevant vtbl entries from the global hash table, and unlinks the object module from the address space of the program. The top-most layer consists of class `Type` which provides methods to dynamically link and unlink **E** classes. We have modified the virtual

```
// file MetaClass.h

#include <E/collection.h>
#include <E/dbStrings.h>
#include "Type.h"
dbclass root {
public:
  root(); //constructorconstructor
  virtual ~root();
  virtual root* make(collection<root>*)=0;
};

dbclass MetaClass {
  dbchar* name;
  collection<root> extent;
  root* exemplar;
  Type* tptr;// pointer to the 'Type' object
  int update_extent(); // to support type changes
public:
  MetaClass(dbchar*,root*);
  ~MetaClass();
  root* create_instance();
  static root* create_instance(dbchar* );
};

// file MetaClass.e

persistent collection<MetaClass> meta_collection;

root* MetaClass::create_instance(){
  return exemplar->make(&extent);
}

root* MetaClass::create_instance(dbchar* n){
  collection_scan<MetaClass>mcs(meta_collection);
  MetaClass* mptr;
  while (mptr=mcs.next())
    if (strcmp(mptr->name,n)==0)
      return mptr->create_instance();
  return 0;
}
```

Figure 4-9: Definition of `MetaClass`

```
#include "root.h"
#include "MetaClass.h"
class Exemplar{public:Exemplar(){};};
dbclass A:public root{
public:
  A(){};
  ~A();
  root* make(collection<root>* c){
    return new (*c) A();
  }
};

persistent A exemplar_A();
persistent MetaClass* meta_class_A =
          new (meta_collection) MetaClass("A",&exemplar_A);
```

Figure 4-10: A Persistent Exemplar Idiom

function invocation mechanism of **E** so that if a program attempts to invoke a virtual function on a persistent instance whose type is unknown to the program, the method code of the corresponding class is dynamically loaded into the program through the corresponding Type object. By making the type information pertaining to each user-defined class persistent in the form of a Type instance, both type identity and type persistence are achieved.

We have also designed a MetaClass to facilitate the creation and maintainance of all instances (the *extent*) of a user-defined subclass. Each MetaClass instance has a pointer to a persistent exemplar instance (of the corresponding class) through which new instances of the subclass can be created. The update_extent method of MetaClass provides a framework for *object migration.* That means, whenever a user-defined type gets modified, all the existing instances of that type have to be migrated to the new version of the type. In a general setup, a user can enter a procedure which converts an instance of a class to an instance of a new version of the same class. This procedure can be dynamically linked into the program and then be invoked on the existing instances.

The design of **E** lends itself to easy extension. In particular, the virtual function dispatch mechanism of **E** (which involves a double indirection) makes the dynamic linking and unlinking of a class easier because 'vptrs' are not stored in each individual object. Instead, integer type tags are stored. Whenever an **E** class is loaded into a process (or unloaded), we only need to make a modification to the global hash table.

We have seen that most of the C++-based object-oriented database systems need some form of dynamic loading of method code. We have shown how we can extend *Dld++* to build a dynamic linker for **E**. It is possible to extend *Dld++* to other persistent C++ language environments such as the Objectstore CC [11] and the Texas Persistent Store [16].

Chapter 5

Conclusions

In this thesis, we have presented the design and implementation of a portable and extensible dynamic link editor *Dld++*. We started with the design of a basic version of *Dld++* making use of the BFD libraries. The basic version has a class `Dld` which is the dynamic linker class. This class encapsulates the low-level functionalities provided by the BFD libraries. The class provides a method to dynamically link a relocatable object module or an archive library file to a running process. In the case of a relocatable object module, it is searched for symbol information and all the externally visible symbols are added to the dynamic linker's symbol hash table. The dynamic linker then allocates memory for the text and data sections of the object module on the heap and relocates their contents. In the case of archive libraries, the dynamic linker searches the symbol table of the archive and loads those object modules which resolve an undefined symbol on the symbol table of the linker. In the case of shared object, the linker loads the symbols from the shared object into its hash table, maps the shared object to the address space of the process and relocates the shared object. The basic version can only link in C object files. It cannot handle the initialization of static scope C++ objects.

We extend the basic version of *Dld++* to address the issues of static constructors/destructors and shared libraries. The reason why these features are not included in the basic version is that they are dependent on the particular C++ implementation and the specific operating system environment. We demonstrated how *Dld++* can be extended to handle the static constructors of G++. This basically entails providing a linker callback function which can collect special symbols such as the static constructor and destructor symbols on linked lists. We define a new class `gcc_dld` which is a subclass of class `Dld`. The class `gcc_dld` provides two new methods: one for linking (`gcc_link`) and the other for unlinking (`gcc_unlink`). The dynamic linking method `gcc_link` invokes the `link` method of class `Dld`. It then traverses the linker's list of constructor symbols in the forward order and invokes those functions. The unlinking method `gcc_unlink` traverses the linker's list of destructor symbols in the reverse order and calls the method `unlink` of class `Dld`.

We also demonstrate how *Dld++* can extended to handle the SunOS shared libraries. This entails modification of the constructor of the class `gcc_dld`. The modified constructor examines the program's `__DYNAMIC` structure which contains a list of structures describing the shared objects that are to be linked to executable at program startup time. The constructor locates those shared objects and finds out where they are mapped in the process's address space. It then loads the symbols from those shared objects into the linker's hash table.

We have noted previously that a dynamic linker such as *Dld++* can be very useful in the case of object-oriented database systems where it is sometimes necessary to dynamically bind method code to a running process when the process encounters an object of unknown type on the persistent store. We have shown how we can extend *Dld++* to build dynamic linking capability in the case of a persistent C++ language **E**. This also necessitates some modifications and extensions to the run-time support library of **E**. It would be interesting to provide similar dynamic linking facilities to other object-oriented database systems such as the Objectstore [11] and Texas Persistent System [16].

In summary, we have shown that *Dld++* is a general purpose dynamic linking library which is easily extensible and customizable to a variety of applications and incorporating such dynamic linking capabilities will facilitate rapid prototyping of software applications.

References

[1] A. Biliris, S. Dar, and Narain H. Gehani. Making C++ Objects Persistent: the Hidden Pointers. *Software - Practice and Experience*, 23(12):1285–1303, December 1993.

[2] Roy H. Campbell, Nayeem Islam, David Raila, and Peter Madany. Designing and Implementing Choices: An Object-Oriented System in C++. *Communications of the ACM*, 36(9):117–126, 1993.

[3] Michael J. Carey et al. The EXODUS Extensible DBMS Project: An Overview. In Stan Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases.* Morgan-Kaufman Publishers Inc., 1990.

[4] Steve Chamberlain. *libbfd - The Binary File Descriptor Library (Version 2.5).* Cygnus Support, First edition, April 1994.

[5] Steve Chamberlain and Roland H. Pesch. *Using LD - The GNU Linker - Version 2.* Cygnus Support, January 1994.

[6] James O. Coplien. *Advanced C++ : Programming Styles and Idioms.* Addison-Wesley, 1992.

[7] R.C. Daley and J.B. Dennis. Virtual Memory, Processes, and Sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[8] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[9] Robert A. Gingell, Meng Lee, Xuong T. Dang, and Mary S. Weeks. Shared Libraries in SunOS. In *Proceedings of the Summer Conference*, USENIX Technical Conference, pages 131–145, Phoenix, AZ, Summer 1987. USENIX Association.

[10] Wilson W. Ho and Ronald A. Olsson. An Approach to Genuine Dynamic Linking. *Software - Practice and Experience*, 21(4):375–390, April 1991.

[11] Charles W. Lamb, Gordon Landis, Jack A. Orenstein, and Daniel L. Weinreb. The Objectstore Database System. *Communications of the ACM*, 34(10):50–63, 1993.

[12] Joel E. Richardson and Michael J. Carey. Persistence in the E Language: Issues and Implementation. *Software - Practice and Experience*, 19(12):1115–1150, December 1989.

[13] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The Design of the E Programming Language. *ACM Transactions on Programming Languages and Systems*, 15(3):494–534, 1993.

[14] Marc Sabatella. Issues in Shared Libraries Design. In *Proceedings of the Summer Conference*, USENIX Technical Conference, pages 11–23, Anaheim, CA, Summer 1990. USENIX Association.

[15] Donn Seeley. Shared Libraries as Objects. In *Proceedings of the Summer Conference*, USENIX Technical Conference, pages 25–37, Anaheim, CA, Summer 1990. USENIX Association.

[16] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An Efficient, Portable Persistent Store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992. Morgan-Kaufman Publishers Inc.

[17] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Inc., September 1994.

[18] Richard M. Stallman and Roland H. Pesch. *Debugging with GDB - The GNU Source-Level Debugger.* Free Software Foundation, Inc., 4.12 edition, January 1994.

[19] Bjarne Stroustrup. Possible directions for C++. In Jim Waldo, editor, *The Evolution of C++ : Language Design in the Marketplace of Ideas*, pages 53–73. The MIT Press, 1993.

[20] Sun Microsystems. *Programming Utilities and Libraries*, March 1990.