A Distributed Version of the C Programming Language

Mike Crowe
Department of Computer Science
University of Warwick, UK
<mac@fysh.org>

Supervisor: Dr Roger Packwood
1996-1997

Copy One

April 30, 1997

Abstract

This is an investigative project which set out to discover a new way of providing distributed processing capabilities to the C language. It centred on the viability of treating a function as a unit of code that can be executed remotely. The resulting language, *Distributed C*, proves that this approach to distributed programming works.

This document describes the design of the language, a compiler and runtime environment for executing *Distributed C* programs across a network of UNIX workstations. Testing shows that the system can yield great reductions in processing time for some problems.

Keywords

C, Parallel, Concurrent, Distributed,
Sockets, Compiler, 'Dynamic Libraries'

Contents

Chapter 1

Introduction

1.1 What is a distributed system?

A distributed system is a number of interconnected machines working together to perform a task. The workload is shared between them and operations may be executed in parallel to increase performance.

With the decentralisation of computer power there are now many networked machines sat on people's desks sharing resources such as the file store. These machines sit idle most of the time and are therefore not being used to their full potential. Distributed programming allows these machines to be used remotely as part of a larger virtual machine.

Traditionally writing a program that distributes itself over many machines requires the programmer to write special code to perform message passing and remote execution. There are powerful libraries such as *PVM (Parallel Virtual Machine)*[1] and *MPI (Message Passing Interface)*[2] for doing this.

1.2 How does Distributed C help?

The purpose of *Distributed C* is to make it easier for a sequential program written in the C Programming Language to be modified to use the power of many workstations at once.

Programmers already have a large knowledge of C so as an extension to the C language *Distributed C* can quickly be understood and applied.

The unit of distribution in *Distributed C* is the function. A function may be executed on a remote host with the parameters and return value automatically sent between the machines involved. No other types of communication are provided for in the language. This means that the programmer must modularise code to be executed remotely into a function with a well defined interface. The organisation of code in this way can lead to greater reliability than if messages are being sent around the system in an ad-hoc manner.

Since the distributed features are handled by the compiler, it can check for consistency and ensure that types match between the caller and the remote function. Type safety is a major problem with library based systems since they have to allow transmission of arbitrary

[1] PVM creates a powerful virtual machine using a dynamic pool of hosts. Tasks running on the hosts may communicate with each other using simple constructs.

[2] MPI provides many ways to allow tasks to communicate between machines of the same architecture

types. The compiler can also ensure that pointers to data that will be inaccessible on a remote host cannot be sent.

1.3 What is Distributed C?

Distributed C is a superset of the C programming language suitable for distributed programming. It adds a small number of keywords to the language to do this. The *Distributed C* system consists of two parts: a compiler and a runtime system. The compiler accepts *Distributed C* source code and outputs C code suitable for compiling with *gcc* into a native code shared object file. The runtime system then executes the program using the shared object file, possibly on many machines.

Figure 1.1 shows an example *Distributed C* program, it does not actually perform any useful work but it serves as an example of the new features.

The '`remote`' keyword indicates that the function declared following it can be executed remotely.[3] The `add` function it declares is used to add two numbers together. The `main` function declares two '`process`' variables which are used to represent processes being run. Two remote processes are started by calling the `add` function preceded by the `dstart` keyword. Once the processes are started the two variables `pa` and `pb` can be used to refer to them. The `while` loop waits for both of the processes to complete and then exits. The `dselect` operation waits for one of the processes it is passed to finish and returns its index (it is also possible to pass arrays of processes). Once the index is known the `dwait` operation is used to retrieve the actual result of the function. The `dselect` operation ignores a process variable if its result has already been retrieved.

The example program can be compiled into C using the *Distributed C* compiler driver program `dcc`, and then compiled to native code as follows:

`dcc -d example.dc` yields a C file called `example.gen.c`
`dcc example.gen.c` yields a shared object file called `example.so`

These two operations can be combined into:

`dcc example.dc` yields a shared object file called `example.so`

Both methods generate a shared object file called `example.so`.

Since the program is to be run across many machines it is necessary to tell the runtime system the names of these machines. It is also possible to compile the system to run concurrently on a single machine, possibly taking advantage of multiple processors. This is done by entering their names, one per line, in a simple text file called `example.hosts`.[4] For this example only two remote processes are started so any more than two machines in the file would be redundant. The system requires daemons to be running on each machine so that they can execute functions. They can be started manually using '`rsh`' or the daemon itself can be used to start them automatically. The command

```
dcd -r example.so
```

will start the daemons on the machines. It is now a good idea to check that they are running correctly. This can be done with the command

```
dcd -p example.so
```

[3]From a language point of view this is not strictly necessary since this information could be determined when the function is called in a distributed manner. However, it does make the parsing and code generation easier.

[4]The name is generate from the shared object filename.

```
#include <stdio.h>

remote double add(double, double);

int main()
{
    process<double> pa,pb;
    double ra, rb;
    int i, j;

    printf("Starting processes\n");

    pa = dstart add(3.0, 7.2);
    pb = dstart add(4.7, 12.23);

    i = 2;
    while (i)
    {
        j = dselect(pa, pb);
        printf("Return from dselect = %d\n", j);
        switch (j)
        {
        case 0:
            ra = dwait(pa);
            printf("Result from A = %f\n", ra);
            --i;
            break;
        case 1:
            rb = dwait(pb);
            printf("Result from B = %f\n", rb);
            --i;
            break;
        default:
            printf("Something odd happened, dselect returned %d\n", j);
        }
    }
    printf("Everything is finished\n");
    return 0;
}

remote double add(double a, double b)
{
    return (a + b);
}
```

Figure 1.1: *An example Distributed C program*

---

which will ‘ping’ each daemon to check that it is responding correctly. Now the program can be run as many times as required by issuing the command

`dcd -m example.so`

at the shell prompt. Once the program is no longer required it is advisable to stop the daemons on the remote machines using the command

```
dcd -k example.so
```

1.4 Techniques

The compiler and runtime system make use of a number of tools.

- Concurrency within the runtime system is handled using the POSIX threads library.
- Communication between hosts is performed using BSD sockets.
- The *Distributed C* source code is compiled into a shared object file which is treated like a dynamic library. This means that it can be opened and arbitrary functions executed from inside it. The shared object file also calls functions in the daemon.
- The lexer and parser are generated using *Flex* and *Bison*.
- The whole project is made portable (between at least Solaris and Linux) through the use of GNU Autoconf.
- The compiler is implemented using object oriented techniques in C++.

Chapter 2

Research

The original idea for the project came from my supervisor who suggested that an appropriate boundary at which to implement a distributed system would be a function call. I originally considered that it would be possible to take a C program and analyse the function calls to allow the call to be made a while before the answer is required (a 'future'). This method means that a function call could be pushed back in the code to as early as possible and the return value only actually waited for when it is required.

I built up ideas for implementing this method but was eventually persuaded that explicit parallelism would be more flexible and also easier to implement consistently. This led me to create a system which allows remote processes to be started asynchronously and its results later retrieved.

2.1 Language

2.1.1 ParC

In the words of its authors "*ParC* is a block-oriented, shared memory, parallel version of the popular C programming language. *ParC* presents the programmer with a model of a machine in which there are many processors executing in parallel and with access to both private and shared variables." [2]

The programmer uses explicit parallelism based on the block structures added to the language – there are three new block types added, two `for` loops and a parallel block.

The language was designed at Jerusalem University, Israel.

Parfor loop

A `parfor` loop is similar to a normal `for` loop but it executes the body of the loop in parallel the correct number of times. The parameters are radically different from a standard `for` loop: the index variable, `ivar`, is implicitly declared as an integer and it ranges from `start` to `stop` in increments of size `step`.

```
parfor(ivar; start; stop; step)
   stmt
```

Lparfor loop

It is inefficient for a `parfor` loop to execute many iterations when there are few processors since time is wasted with context switches as each competes for processor time. The `lparfor` loop splits the number of iterations into sets to be executed by each processor, the code within each set is executed serially. This method makes better use of processor time. Figure 2.1 shows how such code may be executed.

Example
```
void operate(int);
lparfor(i;1;9;1) { operate(i); }
```

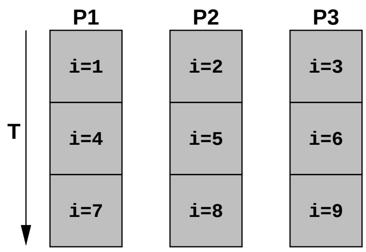


Figure 2.1: *The execution of a* `lparfor` *loop across three processors.*

Parblock

A parallel block allows arbitrary code to be executed in parallel. At the end of the block the program only continues when all the activities have completed. The statements within a `parblock` are separated by two colons.

```
parblock {
   stmt-list
::
   stmt-list
::
   stmt-list
}
```

Variables

Automatic variables declared inside an activity block are completely local to the activity. In the parallel `for` statements a private instance of each variable will be created for each activity. This behaviour can be circumvented by declaring the variable as `static`, this ensures there will only be one instance of the variable — this is analogous to the normal meaning of `static` in a C program. Since variables may be shared there is an atomic *fetch-and-add* operation provided called `faa`, according to the authors this has proved useful in implementing parallel algorithms. The language also provides facilities for storing variables close to appropriate processors in multiprocessor machines without shared memory.

Extra facilities

A code block that is executing as part of a parallel activity cannot be killed since there is no way of identifying it. However, a block can terminate itself, descendants and siblings.

`pcontinue`	Stop the current activity.
`pbreak`	Stop this activity, all its siblings and descendants.
`return`	Terminates all activities generated within the function.

Figure 2.2 shows an Example *ParC* program.

```
parfor(i; 0; N-1; 1) {
    static int s; /* Only one s for ALL iterations */
    int t;        /* One t for EACH iteration */
    faa(&s, 1);   /* Add one to s safely */
    t++;          /* Own instance of t means this is safe */
}
```

Figure 2.2: *An example ParC program showing the use of* `static` *variables and the atomic* fetch-and-add *operation. The loop executes* `N` *times.*

This language is far more complex than anything I could achieve in *Distributed C*. It has undergone a few years of development by a group of people. Also, the language is not distributed across machines so pointers and global variable access are available.

2.1.2 Concurrent C

Concurrent C is a large extension to the C language developed at AT&T Bell Laboratories, [6]. It is a large language that supports concurrent (but not distributed) execution of C code. Processes are declared in much the same way as functions but they are invoked using a different syntax. Once a process is running it may take part in 'transactions' which appear to be like function calls but the remote process accepts them in a explicitly. Figure 2.3 shows an example program. The language also supports guarded select statements and monitors.

The language is implemented as a preprocessor which outputs C code. It can also handle C++ input and generates C++ output.

2.2 Compiler

My main source of information about writing the compiler were *Compilers: Principles, Techniques and Tools* [1], *Lex & Yacc* [11], the `comp.compilers` newsgroup [5] and the Compiler Design lectures [9]. Initially I tried to build a simple parser and associated code which would accept valid ANSI C input and output it again from the parse tree.

Program
```
process spec printer(int id)
{
    trans void print(char *msg);
}

int main()
{
    process printer p0, p1;    /*process ids  */
    p0 = create printer(0);    /*create procs,*/
    p1 = create printer(1);    /*and save ids */
    p0.print("Good Morning!"); /*call trans   */
    p1.print("Good Morning!");
    p0.print("");
    p1.print("");
}

process body printer(id)
{
    for(;;)
        accept print(msg) {
            if (*msg == 0)   /*if msg is "",*/
                break;       /*stop process */
            printf("%d:  %s\n", id, msg);
        }
}
```

Output
```
0:  Good Morning!
1:  Good Morning!
```

Figure 2.3: *An example Concurrent C program taken from 'The Concurrent C Programming Language', [6].*

2.3 Runtime System

2.3.1 Target Platform

UNIX seems to be the most logical platform for the runtime system since it natively supports the communication protocols and tools I will require. UNIX also provides remote login facilities to enable me to test the runtime system from a single workstation or terminal. However, I intend to minimise the platform dependencies so that the system could be ported to other operating systems in the future. In particular, *Microsoft Windows NT* would seem a suitable platform.

The equipment available in the Computer Science department runs a mixture of *SunOS* and *Solaris* (both supplied by Sun Microsystems), with *SunOS* gradually being phased out. To simplify demonstrations and offer maximum security in the event of equipment failure it is important to me that the system runs on departmental equipment – this effectively means *Solaris*.

To enable me to develop the system on my own machine it would also be useful if the system would run on an Intel based PC running *Linux*.

If there is enough time available Computer Services provides access to a Silicon Graphics IRIX machine for computationally intensive tasks. It may be possible to run the system on that platform.

So, in summary the runtime system should run on Solaris and Linux operating systems. Although I will only be able to test it on SPARC-Solaris and i386-Linux there should be no reason why it won't work on x86-Solaris, SPARC-Linux, Alpha-Linux and other variants provided the necessary libraries are available.

2.3.2 Available Technologies

There are a number of recent enhancements and external libraries available for the various variants of the UNIX operating system that could be relavent to this project. I needed to investigate each of these so that I could make informed decisions during the design of the project about what was available.

Threads

There are many threading libraries available for various versions of UNIX, they can be split into user-space and kernel-space implementations. Figure 2.4 shows the issues involved in the different implementations. From this information I decided that kernel-space threads best suited my application, the important factors were the multiple processor capability and operating system support.

The use of kernel-space threads greatly limited my choice of threading libraries. Under Solaris 2.5 two similar libraries were available, POSIX threads and Solaris threads. Under Linux most of the libraries were still in the early stages of development but a POSIX threads implementation called *LinuxThreads* seemed to be the most stable.[1] Using POSIX threads also meant that portability to other UNIX platforms should be easier.

Threads will need to be used inside the runtime system so that a function can be started remotely and its result received whilst the rest of the program continues running.

RPC

RPC was initially implemented by Sun Microsystems as a means of supporting their Network File System. An RPC system is defined as a "collection of software necessary to support remote procedure call programming and the necessary run-time services." [3]. It now forms the basis of many other network systems such as *NIS (Network Information System)*[2].

Table 2.5 shows the widely available RPC implementations. ONC RPC comes as part of the standard operating system on Solaris, Linux and the majority of other UNIX platforms since the source is freely available and RPC support is required for NFS.

RPC, in conjunction with the RPCGEN protocol compiler, allows arbitrary procedures to be executed on a remote host as if they were executed locally. To support this the parameters and return value of the function must be declared in a protocol definition file which is then

[1] In fact, the next version of the standard C library for Linux, `libc6`, will incorporate *LinuxThreads*.

[2] NIS was originally called Yellow Pages.

User-space threads

⊖ Once one thread is blocked inside the kernel all threads inside the process are unable to run. This means that system calls like `select` or `write` must be wrapped to stop them blocking.

⊖ It is difficult to take advantage of multiple processors since the kernel will only assign one processor to a running process.

⊕ Context switching (within the threads of the process) is generally more efficient since there is no system call or kernel overhead.

⊕ User-space implementations are available for most modern versions of UNIX.

Kernel-space threads [10]

⊖ Context switches may be expensive, especially when threads block.

⊖ Thread creation is expensive since the kernel must effectively create a new process.

⊕ Multiple processors are assigned by the kernel, so two threads may *really* be running concurrently.

⊕ Scheduling may be smoother since the same scheduler (the kernel's) handles both threads and other Unix processes.

⊕ Better support (at least under Solaris) since kernel threads are an integral part of the operating system.

Figure 2.4: *Issues involved with the different threads implementations.*

Abbreviation	**Name**	**Supporters**
ONC	Open Network Computing	Sun (source free, available for most platforms.)
NCS/ DCE	Network Computing System Distributed Computing Environment	Apollo, Hewlett Packard, Open Software Foundation, Microsoft

Figure 2.5: *Differing implementations of RPC.*

compiled to form wrappers for the functions. The types available to the protocol compiler are similar to those available in C.

Using RPCGEN for *Distributed C* would require the compiler to output protocol definitions, which would then need to be compiled individually adding great many files and complexity to the build process. I considering using RPC directly rather then through RPCGEN but this seemed to yield no real benefits over using BSD sockets.

PVM/MPI

"PVM [(Parallel Virtual Machine)] is the existing de facto standard for distributed computing and MPI [(Message Passing Interface)] is being touted as the future message passing standard.", [7]. It is important for me to investigate these standards as both a source of ideas for the *Distributed C* language and as a possible method of implementation.

PVM creates a virtual machine consisting of a set of heterogeneous hosts which appear to be a single, large, parallel computer. Its main features are its portability, scalability, fault tolerance and the ability to use machines of different architectures in a single virtual machine. PVM allows tasks to be created which use explicit message-passing constructs to communicate. Since its inception in 1989 PVM has been enhanced through PVM 2 in 1991 to the current PVM 3 in 1993. It has gained the ability to support third party debuggers, facilities for ATM (Asynchronous Transfer Mode) networks and transparent use of shared memory. The PVM virtual machine is dynamic, machines can be added and removed at any time using the system console or under program control.

The MPI standard was specified by high performance computing experts between 1993 and 1994. It provides a large set of point-to-point communication routines and methods to send messages to groups of processes. The source of communication methods is much richer than PVM. For performance reasons MPI does not require support for cross-architecture communication or between programs written in different languages — this effectively means that it does not support such facilities. Again for performance reasons, MPI uses a static host pool.

PVM and MPI can be used together and there is at least one project, called PVMPI, attempting to combine them.

Neither PVM nor MPI provides any support for security. There appears to be no way of stopping a malicious user interacting with the running tasks.

Using either PVM or MPI to implement the runtime system for *Distributed C* would require me to centre the whole system around them. PVM uses a system console which I felt would get in the way and be confusing. Both tools also seem to require a certain amount of knowledge about the available machines. I don't really see that there would be much to gain over implementing my own simple message passing.

Sockets

BSD sockets are the lowest level method of communicating between machines that is still portable. It is an integral part of the TCP/IP protocol that is now supported well beyond UNIX due to the proliferation of 'The Internet'[3]. Each host on a TCP/IP network has at least one IP address which uniquely determines it. Each host notionally has a pool of numbered ports which can be used to provide network services. Many Internet services run on 'well known' port numbers, for example electronic mail is transferred on port 25 and web pages are usually transferred on port 80. The actual choice of port number is arbitrary as long as it is standardised.

A socket connection between server and client is a 'virtual circuit' which behaves as if there is a dedicated link between the two programs even though the data is actually being converted into packets which may follow entirely different routes between the hosts.

[3]BSD sockets are supported on all modern variants on UNIX, VMS, The Apple Macintosh and both 16 and 32 bit versions of Microsoft Windows

Figure 2.6 describes a simple socket communication.

Server	**Client**
create socket bind socket to port 1732 listen to socket	
	create socket connect socket to server port 1732
accept incoming connection	
Server and client now use read and write system calls to communicate.	
close socket	close socket

Figure 2.6: *An example of communication over a socket.*

Socket communication is ideal for executing a remote procedure since the parameters can be sent across and then the client will automatically wait for the reply to be sent.

Dynamic Libraries

In early versions of UNIX (and other operating systems) programs were compiled and linked *statically* with libraries. This lead to relatively small programs (such as `rm` and `ln`) containing duplicate code which took up disk space and memory. Shared libraries (also known as dynamic libraries) provided an alternative to this by placing the entire code for the library in a single 'shared object' file which can then be paged into the memory space of every process that requires it. Shared libraries save both disk space and memory.

Early implementations of shared libraries were difficult to create with extra data structures known as jump tables being required to ensure that the library could be upgraded to a limited extent. The creation of a new binary format called ELF meant that these problems disappeared. Creating an ELF shared library is now very easy.

The ELF system also provides it's own library (called `libdl`) which allows a program to call functions inside a shared object very easily without even knowing the library or function name at compile time.

Both Solaris and Linux allow for ELF executables and libraries. Other operating systems such as SunOS provide the `libdl` library for their executable formats but I did not need to look any deeper.

GNU Autoconf

Writing a program for UNIX that will compile and run its many variants is very difficult. Some degree of portability can be achieved by sticking to the POSIX.1 standard, but many operating systems support far more than the standard and some of this functionality may be useful, or even essential. When it becomes necessary to differentiate between platforms it can sometimes be done by checking preprocessor symbols like `__SVR4__`, but this can cause its own problems since it isn't always clear exactly what the presence or absence of a symbol implies.

Some programs like *PERL (Practical Extraction and Report Language)*[4] provide their own complex configuration programs which analyse the system before compilation. The Free Software Foundation has generalised this concept into Autoconf, a large number of *m4(1)* macros that allow a configuration script to be created. This script provides a large amount of information about available header files, libraries, executable programs, and many other parts of the system.

Using `autoconf` will allow me to ensure that the code is portable between at least a limited number of platforms.

[4]Perl was written by Larry Wall and has gained prominence through its use in CGI scripts for web servers.

Chapter 3

Specification

The specification for the language is deliberately imprecise since it is intended to be an ongoing development with new features inspired by experience with previously developed ones.

3.1 The Problem

As explained in the introduction there are great gains to be made by running programs in parallel across many workstations. Methods exist for doing this such as *PVM (Parallel Virtual Machine)*[1] and some versions of *RPC (Remote Procedure Call)*[2] but they require the programmer to become very involved in the distribution of the program execution. Languages such as Concurrent C and *ParC* allow concurrent processes but their methods do not extend well to distributed programming.

3.2 The Aim

The aim of this project is to create a version of the C programming language that makes it straightforward for programmers to produce distributed programs. The system should just require the names of the machines to use and it will control where the actual code is executed itself. The programmer should not have to spend large amounts of time learning about complex library routines. It should also be possible to compile a sequential, single threaded version of the program from the same source code without alteration.

3.3 The Methods

The project will involve producing:

- a library of routines to provide the distribution function;
- a library to send and receive function call requests;
- a compiler for the language.

[1] a system for passing messages between different *processors* considered to be part of a single virtual machine.
[2] a system for providing platform and operating system independent procedure calls

3.4 Timetable

It would seem sensible to split the work into that necessary to ensure distributed execution and the code to ensure parallelism. Since the second is reliant on the first this dictates the order.

Order of work:

1. Make a compiler that will accept a valid C program and output it again from the parse tree. This will have to correctly parse types.
2. Write a simple library to simulate remote execution and test programs for this that can be used as models for the compiler output.
3. Add language features to the compiler and make sure it outputs the correct code to interface with the library.
4. Write various example programs for testing the system and suitable for use in the demonstration.
5. Implement the distribution in the library.

The design for the library and the compiler will need to cater for concurrent execution although initially it will not be implemented. In reality the first two items are likely to be implemented at the same time.

3.5 Resources

The project files will be stored using the *CVS (Concurrent Versions System)*[3] version control system to allow me to work on both the code and documentation on my own machine and on the departmental machines.

Since this program is intended for use by users no system privileges will be required.

[3] a version control system based on *RCS* released by the Free Software Foundation.

Chapter 4

Design

4.1 First ideas

Since the purpose of this project is to design a distributed version of the C programming language it would seem sensible to avoid changing the actual language too much. Doing this makes it possible to make the distributed C compiler a preprocessor which will simply pass through most of its input. This approach makes the compiler for the language far easier to implement and is the approach taken by many other languages, including some versions of C++. Adding few keywords and avoiding changing the structure of the language means that it should be possible to compile the program as straight C with little modification.[1]

I intend to design the language so that it requires as little programmer intervention as possible. Since the purpose of the language is to execute code on multiple machines it will be necessary to decide which machines run which code – ideally this would be dynamic and take into account processor load but for the purposes of this project it may be necessary to provide this information statically for a single execution of the program.

4.1.1 Distribution Strategy

Making a sequential program work in parallel is not an easy process, working out the interdependencies of a piece of sequential code for parallelising it is a very complex problem. However, in a high level language a function represents a reasonably self contained unit of code. In fact if the function is banned from using static or global data, cannot access external data through pointers, and is only allowed to call functions implemented similarly, it is a completely separate unit that can have no side-effects. A C function restricted like this effectively becomes equivalent to the mathematical definition of a function as used in functional programming languages. A program made of such units could easily be split up into separate components running on different machines without the problems of shared memory and mutual exclusion.

For example, consider a program which calculates the Mandelbrot set and produces a bitmap from it. Whether a point is in the set or not is completely independent from the other points being considered. Figure 4.1 shows a pseudo-code program which calculates a subset of the Mandelbrot set in parallel. The program can take advantage of many processors to perform the iterative mandelbrot calculation in parallel. The information required to perform

[1]Hopefully just a few uses of `#ifdef`.

this calculation is small — the coordinates of the point — and the actual answer is a simple boolean yes or no. It is this type of problem that is ideally suited to distribution across many processors. A program which requires access to large amounts of changing data is not so suitable for distribution since the amount of time spent sending the data between the machines over a slow network connection is high compared to the amount of processing that can be performed.[2]

```
function is_mandelbrot(x, y)
  return true if point (x,y) is in the mandelbrot set.

for each y on the y-axis
  for each x on the x-axis
    start parallel function is_mandelbrot(x, y)

for each y on the y-axis
  for each x on the x-axis
    if result from is_mandelbrot(x, y) is true then
      add (x,y) to set.
```

Figure 4.1: *A pseudo-code parallel Mandelbrot set program.* $x \times y$ *instances of the* is_mandelbrot *function are started in parallel.*

4.2 Assumptions and Simplifications

Clearly a fully capable distributed system is very complex and beyond the scope of a third year project. So, I will have to simplify the problem.

- Firstly, all the machines in the distributed system must be running the same operating system on similar architectures. For example, I will not be converting data between different byte orderings or floating point formats.
- The machines must share a commonly mounted file system. This is required so that a filename can be passed between machines and still be valid. This assumption is probably stronger than it needs to be – with a bit of care it may be possible to use completely separate systems but the programmer will have to bear this in mind when performing file access.
- I will rely on the existence of reliable communications between machines, even though the machines may be many miles apart – I do not intend to provide any facilities for fault tolerance.
- I need to rely on the programmer not to try and subvert the system. Eventually warnings could be provided during compilation to warn the programmer that they are invoking undefined behaviour.

[2]This may change with the advent of high speed ATM (Asynchronous Transfer Mode) networks and clusters of machines again becoming popular.

- Initially, at least, I will not be providing any support for security. Obviously in a final implementation remote machines will need to authenticate requests to execute functions on a host.

- Since there is no shared memory space it will not be possible to pass pointers between machines - ideally the compiler should not accept such programs.

  Initially this may seem very limiting for a language based on C that is built around pointers. However, since arbitrary structures can be passed I believe that this will not prove to be too much of a problem. For example, a string can be passed by declaring a structure:

  Example
  ```
  struct lstring {
      char s[256];
  }
  remote struct lstring rrev(struct lstring);
  ```

  However, this method means that the string length is limited in this case to 256 characters (the limit is arbitrary, but it must exist). It may be possible to relax this rule by allowing C style string pointers and providing special packaging support.

- Finally, I will need to make a number of small simplifications to the C language to ease parsing, in practise very few programs use these features. These are explained in Section 5.2.1.

4.3 Language Design

At the time of the oral presentation I was considering two different approaches to the language design:

- **Process type variables**
  Keywords to allow starting, querying and retrieving results from remote processes. This method required a (potentially) running process to be represented by a type. This follows the fork/join model of concurrent programming.

- **Block structured distribution**
  Distributed versions of C control-flow structures, e.g. a distributed `for` loop.

Although the actual syntax changed during development the basic design has stayed the same.

4.3.1 Process type variables

This method involves the introduction of a new type. A process variable can be used to represent a process which consists of a remote function returning a specific type. Extra keywords are used to control processes represented by these variables.

Description

- **Declaration**
  The declaration of a process variable is reminiscent of C++ template instantiation and performs a similar task.

  In this example the remote function must return an integer[3]:

  Example `process<int> p;`

- **Starting distributed processes**
  A function can be started remotely using the `dstart` keyword. Its purpose is to start the function executing on a remote machine and associate it with a process variable, in this case `p`.

  In this example the function `remote_proc` is started remotely and the variable `p` can be used to refer to it.

  Example `p = dstart remote_proc(12.0);`

- **Retrieving return values**
  Once a function has been started at some point its return value will need to be retrieved, this is achieved using the `dwait` keyword[4] which yields the return value of the remote function if it is available or waits until it is available.

  This example declares the variable `p` to be a process returning `int`, starts a remote process then waits for it to complete.

  Example
  ```
  process<int> p; int a;
  p = dstart remote_proc(12.0);
  a = dwait(p);
  ```

- **Waiting for many processes to complete**
  It is also possible to wait for one of many remote functions to complete using the `dselect` keyword. The syntax and semantics are more complex than the other keywords due to the large amount of functionality provided. The selection process is analogous to the UNIX `select` system call.

  The keyword is followed by any number of parameters except zero. Each parameter can be a simple process variable or a process array variable and bound.

  Example
  ```
  process<int> q[10]; int n, a;
  n = dselect(q:10);
  a = dwait(q[n]);
  ```

- **Terminating processes**
  Remote functions can also be explicitly terminated if their result is no longer required using the `dkill` keyword. If the process has already completed or has already been terminated then no action is taken. Many processes may be specified in a similar manner to `dselect`.

  Example
  ```
  dkill(p);
  dkill(q:10);
  ```

[3] The original syntax was: `int process p`

[4] This keyword was originally `dresult` but during implementation became `dwait`. With hindsight the original name was probably more logical.

Backus Naur Form

The BNF for each of these keywords is simple to integrate with the rest of the C grammar. Symbolic names not defined in Table 4.2 (e.g. *type_name*) are defined as part of the C grammar and should be self explanatory.

process_specifier	→	**process** < *type_name* >
dc_selection_arg	→	*assignment_expression*
		$\mid$ *assignment_expression* **:** *constant_expression*
dc_selection_list	→	*dc_selection_arg*
		$\mid$ *dc_selection_list* **,** *dc_selection_arg*
dc_expression	→	**dstart** *identifier* **(** *argument_expression_list* **)**
		$\mid$ **dwait** **(** *expression* **)**
		$\mid$ **dselect** **(** *dc_selection_list* **)**
dc_statement	→	**dkill** **(** *dc_selection_list* **)** **;**

Figure 4.2: *BNF grammar for the process variable syntax.*

4.3.2 Block structured distribution

A block structured approach to describing the parallelism is more in keeping with structured programming, but is of course less flexible.

Description

- **Distributed block**
  A distributed block is a C block preceded by the keyword `dblock`. The block can only contain remote function invocations with an optional assignment.

  This example starts the two remote functions `remote_proc1` and `remote_proc2` in parallel and then waits for them both to complete.

  Example
  ```
  int a, b;
  dblock {
      a = remote_proc1(12.0);
      b = remote_proc2(3.0);
  }
  ```

- **Distributed for**
  A distributed for loop is a greatly simplified form of the standard C for loop which may contain only one statement which must be a remote function invocation.

  This example starts ten copies of the `remote_proc` function in parallel and then waits for them to complete.

  Example
  ```
  int i, a[10];
  dfor(i = 0; i < 10; i++)
      a[i] = remote_proc(i*1.2);
  ```

- **Distributed logical operations**
  In standard C the logical operators `&&` and `||` are defined to short circuit when possible. This feature can be used to improve efficiency and also to simplify programs. For example: `if (p && p->count > 0) p->count++;` ensures that p is non-*NULL* before using it.

  The *Distributed C* keywords `dand` and `dor` also short circuit. The keywords are used like a function that can take any positive number of arguments. All the processes are started at once and if the results of finished processes are sufficient to give the result then the program continues execution disregarding the remaining results.

  Example
  ```
  if (dand(remote_proc1(12.0), remote_proc2(6.3)))
      printf("It's true");
  ```

Backus Naur Form

The BNF for this again fits in easily to the C grammar, it is detailed in Table 4.3.

dc_opt_assign	$\rightarrow$	ε | *unary_expression* =
dc_invoke	$\rightarrow$	*dc_opt_assign identifier* (*argument_expression_list*) ;
dc_invoke_list	$\rightarrow$	*dc_invoke*
		| *dc_invoke_list dc_invoke*
dc_statement	$\rightarrow$	**dblock** { *dc_invoke_list* }
		| **dfor** (*expression_statement expression_statement expression*) *dc_invoke*
		| **dfor** (*expression_statement expression_statement*) *dc_invoke*

Figure 4.3: *BNF for the block structured distribution syntax.*

4.4 Compiler Design

4.4.1 Overview

The compiler converts *Distributed C* source code to real C code so that it can be compiled for the required platform and executed. *Distributed C* programs are not real executables but shared libraries (see Section 2.3.2) which require the runtime system to execute.

The compiler is written in C++ using *Flex* and *Bison* compiler generation tools. Most of the parsed source code is passed straight through the compiler to the output file since it requires no changes. The new constructs are translated into appropriate implementation code and type declarations are augmented with extra declarations to allow them to be passed to other machines.

All types are parsed into a canonical form by the compiler, this allows type checking to be performed.

4.4.2 Implementation Language

It is now usual to use a compiler generator rather than write a compiler by hand. The most common tools are *Lex*, which produces the tokeniser, and *Yacc* (Yet Another Compiler Compiler), which produces the parser. *Lex* produces a deterministic finite state automata (DFA) and *Yacc* produces a LALR(1) (One token Look-Ahead Left-to-Right) grammar.

The Free Software Foundation has developed a greatly improved version of *Yacc* called *Bison* and also distributes an improved version of *Lex* called *Flex*. I will use these freely available tools in preference to the originals which suffer from a large amount of variation among different platforms.

Flex and *Bison* generate C or C++ code — this effectively limits my implementation language for the compiler. I have decided to use C++ since it is the language I have most experience with and I also think that the components of a compiler map to objects very well.

4.4.3 High Level Design

The compiler has three major parts: the lexical analyser, the parser and the code generator. Figure 4.4 shows the units of information that are used to communicate between them.

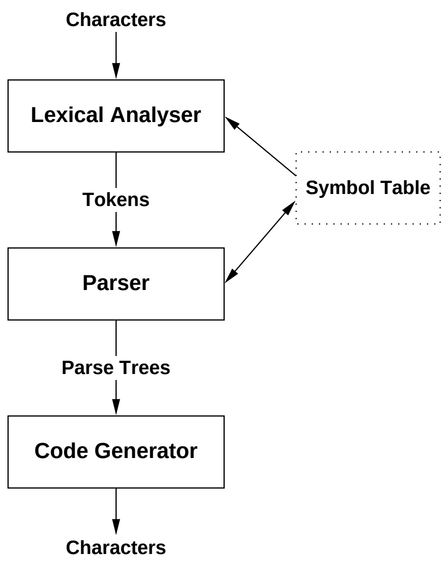


Figure 4.4: *The three major parts of the compiler and the symbol table.*

The compiler will actually need to process very little of the C code that it is given. The majority of the source code will simply be written straight to the output file unchanged (apart from the white-space which is removed during lexical analysis).

The majority of the tokens passed from the lexical analyser (lexer) to the parser are of type `TextToken` or `CharToken`. These types can be used to reconstitute the original text that was input.

Types are created in the lexer for each token in the input that can represent a type (including `typedefs`). When these types are parsed they are combined with a `compose` operation. For example, two type objects representing `short` and `int` are combined to represent `short` `int` in a single type object. Signs (`unsigned`, `signed`), qualifiers (`const`, `volatile`) and storage classes (`extern`, `static`, etc.) are handled similarly.

A complex type such as '`const char *[20]`' is represented as a linked list of types: '`ArrayType(20)` → `PointerType` → `BasicType(const, char)`'. Every type can have any combination of qualifiers (`const`, `volatile` and `restrict`[5]) associated with it, these are handled by the `Qualifiers` class.

Whenever a new type is declared through `struct`, `union` or `typedef` a structure named '`_dcv_`*(mangled type name)*' is declared. This structure contains a special header to allow the type to be used as a return value for a remote function safely. The structure is the actual type of a `process<T>` type in the generated code. The reasons for this are described in Section 4.5.2.

Once inside the parser the majority of tokens are just formed into an ordered list owned by a `CompoundToken`, these in turn are owned by `CompoundToken` objects, thus forming a parse tree. Tokens that need to be treated specially in the conversion from *Distributed C* to C are modified and 'composed' (see Section 4.4.5). Many forms which are parsed hierarchically are also converted to lists, for example function arguments. All modifications to the parse tree are performed as it is parsed.

Once a complete external declaration[6] has been parsed it is written to the output file.

4.4.4 Object Design

There are three different types of object in the compiler:

1. **Token objects** are used to represent parts of the source code. This type of object also includes the pseudo-token objects used to represent types, qualifiers and storage classes.

2. **Database objects** maintain information about the source code that covers many input tokens. An example is the symbol table.

3. **Worker objects** have only a single instance and contain a major part of the execution of the program. An example is the lexer object.

Figures 4.5 and 4.6 show how these objects relate to each other using the Booch notation, [4]. Appendix A contains a full description of each object.

4.4.5 Composition

The composition of types is an important part of the parsing process. When a type is parsed it is read as many separate parts which must be joined together to form the actual type.

Types are passed from the lexer into the parser as pointers to generic `Type` objects. When parts of a type are specified together, e.g. '`unsigned short`', the two separate `Type` objects eventually reach a rule in the *Yacc* source file which requires them to be merged into a single `Type` object. The merging of types or 'composition' is performed by one of the various virtual composition functions, in this case `BasicType::ComposeSelf`. The composition in this case

[5]The `restrict` keyword is a new addition to C9X which, in simple terms, allows functions to assume that pointers do not overlap. This has been done to try and match the performance of *Fortran* in mathematical applications.

[6]An 'external declaration' is the basic unit of the source code that can be repeated any number of times to form the translation unit. Function declarations and definitions are external declarations.

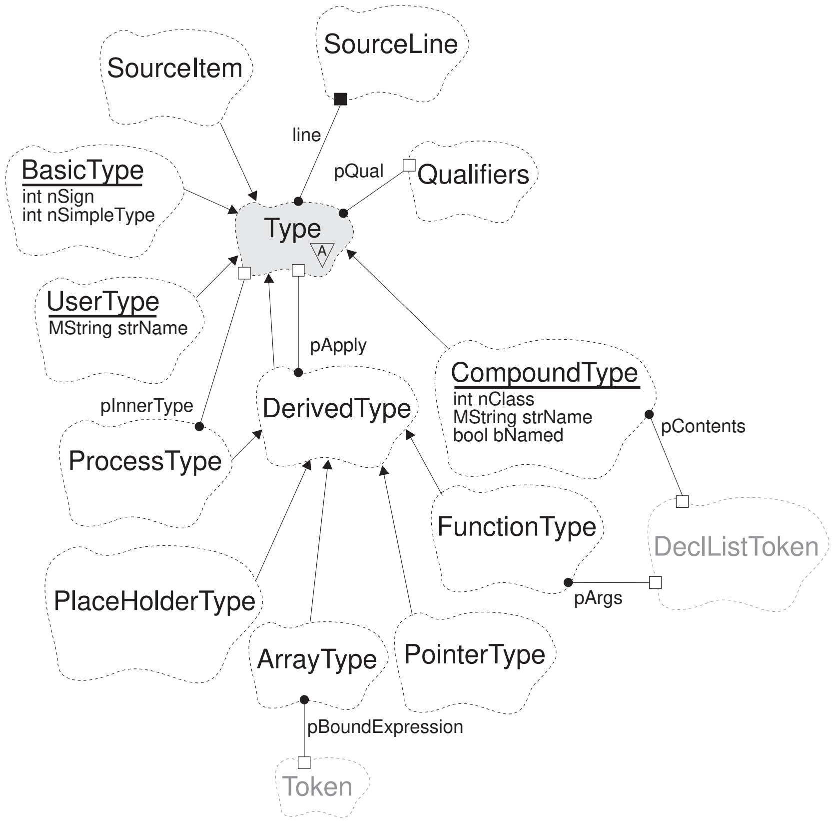


Figure 4.5: *Type classes and their relationships using the Booch notation.*

---

is a simple matter of altering the bit flags in one of the `Type` objects to reflect the joining of the two types and discarding the remaining one. The composition of more complex types such as '`char *`' is slightly more complex although it works on the same principle.

Storage class specifiers and qualifiers can also be composed, this is a straightforward 'bit-wise and' operation on the bit sets used to represent these constructs. Both the `StorageClass` and `Qualifiers` objects provide `ComposeSelf` functions for composing with other objects of the same type. Since a type contains qualifiers part of the `ComposeSelf` function for types invokes the function for qualifiers.

There are two problems with this approach to composition:

1. Most of the operations on types (and tokens) are implemented as virtual functions which each take the appropriate action depending on the actual type of the object rather than the type of the pointer used to perform the operation. For example, this allows the `BasicType::Render` function to correctly build the string determining the type even if it is actually being referred to by a generic `Type` pointer inside a `PointerType` object.

   The composition of two types usually needs to know the types of *both* objects it is dealing with. There is no feature in C++ that directly supports this behaviour[7] so it

[7] A feature has been proposed called 'multi-methods' which appears in a number of other languages but it

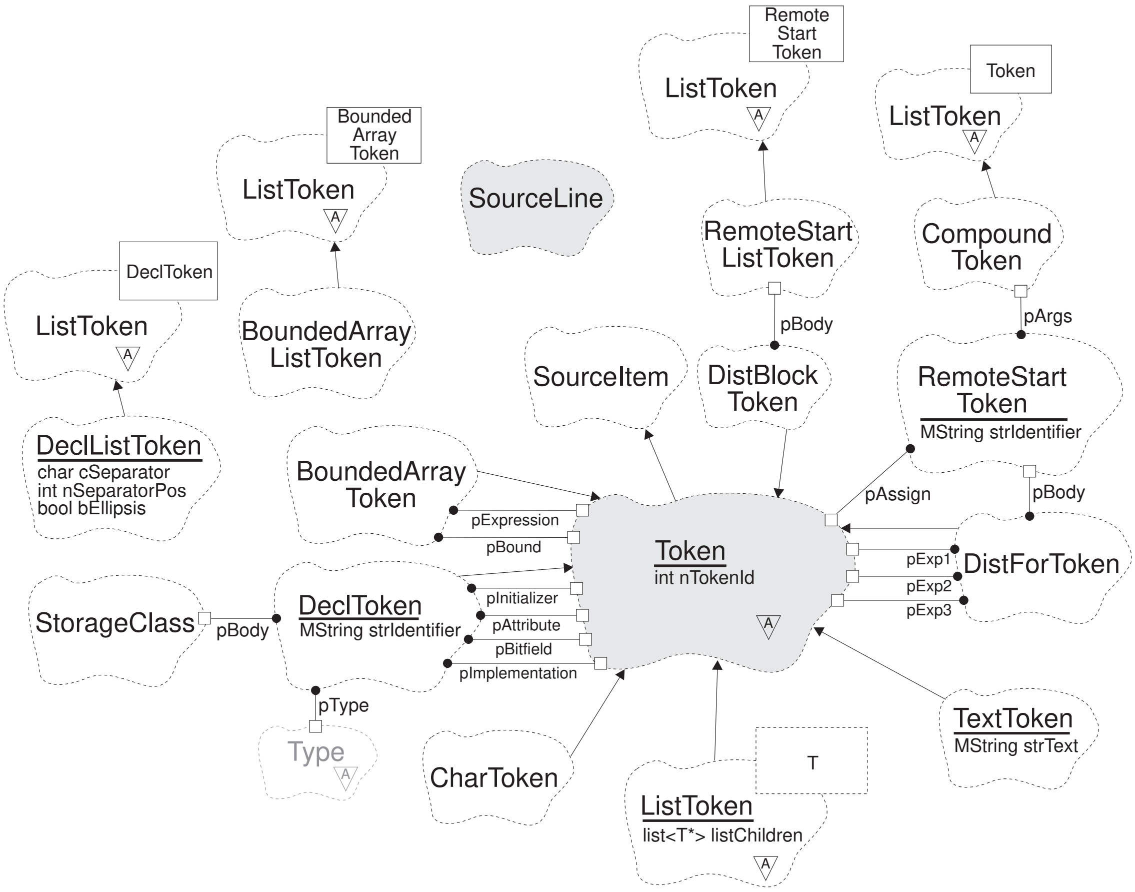


Figure 4.6: *Token classes and their relationships using the Booch notation. To avoid making the diagram over-complex instances of the* `SourceLine` *class are not shown.* `TextToken`, `CharToken`, `DeclToken`, `BoundedArrayToken`, `RemoteStartToken`, `DistForToken` *and* `DistBlockToken` *each contain a* `SourceLine` *object called* `line`.

---

must be implemented using virtual functions and some trickery, one method used is described in *The C++ Programming Language*, [15].

The method actually used was a variant of the Stroustrup method. Most types cannot be composed with most other types, e.g. a `UserType` with a `CompoundType`. Types deriving from `DerivedType` can only compose with types in one direction, a 'pointer to a `char`' is a valid type but a '`char` of a pointer' isn't. These rules can be simplified to those specified in Figure 4.7.

The `BasicType` composed with `BasicType` case is the only time that the exact type of the object is required to perform the composition. This event is handled in `BasicType::ComposeSelf(Type *)`. Ideally it should be possible to use the C++ `dynamic_cast`

was rejected by the committee, [16].

$$
\begin{aligned}
\texttt{BasicType} \circ \texttt{BasicType} &= \texttt{BasicType} \\
\texttt{BasicType} \circ \texttt{Type} &= \text{Illegal} \\
\texttt{CompoundType} \circ \texttt{Type} &= \text{Illegal} \\
\texttt{UserType} \circ \texttt{Type} &= \text{Illegal} \\
\texttt{ProcessType} \circ \texttt{Type} &= \text{Illegal} \\
\texttt{DerivedType} \circ \texttt{Type} &= \texttt{DerivedType} \text{ or Illegal}
\end{aligned}
$$

Figure 4.7: *The composition rules for types, the symbol '∘' is used to represent composition. Whenever a class name is used any of its derived classes may applied equally unless another rule refers to a closer base class. Note that composition is not commutative.*

---

operator to confirm that the second parameter is actually of type '`BasicType *`',[8] but unfortunately the support that *gcc* gives to RTTI is rather incomplete. Instead the virtual function `Type::GetBasicType` provides similar functionality. The `BasicType::ComposeSelf` function calls `GetBasicType` to determine if the object is actually another `BasicType` and if so it calls `BasicType::InternalComposeSelf` which performs the actual composition.

Since a type that derives from `DerivedType` is only the first in a chain it may or may not be possible to compose with it. If an attempt is made to compose with a type derived from `DerivedType` the request is passed on down the chain until an object that does not derive from `DerivedType` is found or the end of the chain is found. If a different type is found then the `ComposeSelf` function will fail correctly (e.g. '`PointerType` → `BasicType`' ∘ '`UserType`' is illegal). If the end of the chain is found then the new type will be attached to it (e.g. '`PointerType`' ∘'`UserType` = '`PointerType` → `UserType`).

2. It is sometimes necessary to promote a `Type` or `StorageClass` object to being a complete `DeclToken` before all compositions are complete, for example, in multiple declarations. This means that composition can still occur inside a `DeclToken`. The member functions `ComposeType`, `ComposeStorageClass` and `ComposeQualifiers` (which forms a simple shortcut) are provided to support this. The `DeclToken::ComposeSelf` function composes with another `DeclToken` object and is implemented by composing the types and storage classes.

The composition functions for all classes are contained in the source file `compose.C`.

4.4.6 Code Generation

There are two approaches to the code generation used within the compiler. The early parts of the compiler (`dwait`, and `remote` function declarations/definitions) perform code generation

[8]The `dynamic_cast` operator uses the runtime type information recently added to the standard to turn a pointer to a base class `A` into a pointer to a derived class `D` only if the actual type of the class is derived from or equal to `D`. If not then the case gives a *NULL* result. For example, consider the declarations:

```
class A {};  class B : public A {};  class C : public B {};
B b;  C c;  A *pb = &b;  A *pc = &c;
```

Then, `dynamic_cast<B *>(pc)` yields a pointer to `c` of type '`C *`' but `dynamic_cast<C *>(pb)` yields *NULL*.

when the constructs are parsed. They generate real tokens to represent the C code necessary to support the *Distributed C* features which are output just like other tokens that are passed straight through. Later parts of the compiler (parts of `dselect`, `dstart`, `dfor` and `dblock`) use special tokens to represent the new forms and the implementation code is generated inside the `GetText` functions. The `BoundedArrayToken`, `DistForToken`, `DistBlockToken` and `RemoteStartToken` work in this way.

The reason for choosing the first method initially was so that a complete tree of valid C tokens existed, this means that at a later date a C compiler back-end could be bolted straight on. This would remove the inefficient conversion from tokens to a text file, by the *Distributed C* compiler, and then back to tokens again by the C compiler. This approach was used by some distributors of C++ to speed up compilation.

The second method gives greater simplicity to the parsing code. The `GetText` functions are generally not complex either since they only deal with an output string rather than many pointers to parts of a tree. A comparison of the `DeclToken::BuildRemotePackFunction` function in `codegen.C` with the `DistForToken::GetText` function in `token.C` shows that the `DeclToken` function is far more complex than the `DistForToken` function.

The code generation of the compiler could be greatly simplified by converting the existing code using the first method to the second method. In fact, the handling of `dstart` was changed to the second method to fit in better with `dblock`. This is not a large task but it would not add any extra functionality to an already working compiler.

Type declarations

Each time a type declaration (`struct`, `union`, `typedef`) is parsed a structure is generated to allow it to be used as the return value of a remote function. This structure is named using the mangled form of the type name. Type mangling is performed by all the type objects representing a type in the parse tree. Each type contributes a number of characters to a name which uniquely identifies the type in a repeatable way. The actual mangling is handled by the `Mangle` virtual function in every `Type` derived object. The actual code generated for a new type follows:

```
struct _dcv_<mangled-type-name> {
    struct _dch_var hdr;
    struct lstring val;
};
```

Remote function declarations

A remote function declaration (this stage is also performed for a function definition which hasn't been preceded by a declaration) generates a structure and an inline function in the output file in addition to the declaration itself with the `remote` keyword removed. The structure is used to hold the parameters for the remote function. This inline function performs the packaging of the parameters, calls the runtime system to start the function remotely and then unpackages the result. The actual code generated follows:

```
struct _dcp_<function-name> {
    struct _dch_invoke hdr;
    (type-of-parameter-1) p_aa;
    ...
    (type-of-parameter-n) p_xx;
};
static __inline__ struct _dcv_(function-name) *
_dcf_(function-name)((type-of-parameter-1) p_aa, ..., (type-of-parameter-n) p_xx) {
    struct _dcp_(function-name) *_dct_p =
        _dcx_allocargs(sizeof(struct _dcp_(function-name)));
    struct _dcv_(mangled-type-name) *_dct_r =
        _dcx_allocvar(sizeof(struct _dcv_(mangled-type-name)));
    _dct_p->p_aa = p_aa;
    ...
    _dct_p->p_zz = p_zz;
    _dcx_setfn((struct _dch_invoke *) _dct_p, "_dcu_(function-name)");
    _dcx_exec((char *) 0, (struct _dch_var *) _dct_r,
              (struct _dch_invoke *) _dct_p);
    return _dct_r;
}
```

Remote function definitions

A remote function definition causes the unpackaging function to be output in addition to the actual function body. The unpackaging function uses the structure declared for the function declaration. The code follows:

```
int _dcu_(function-name)(struct _dch_invoke (*_dct_pi),
                         struct _dch_var (*_dct_pv))
{
    struct _dcp_(function-name) (*_dct_pp) =
        (struct _dcp_bing *) _dct_pi;
    struct _dcv_(mangled-type-name) (*_dct_pr) =
        (struct _dcv_(mangled-type-name) *) _dct_pv;
    _dct_pr->val = (function-name)(_dct_pp->p_aa, ..., _dct_pp->p_zz);
    return 0;
}
```

Remote function invocation

The code generated for a remote function invocation using `dstart` is very simple. This is due to most of the work being done in the inline function. The code generated for invoking a function follows:

```
_dcf_(function-name)((parameters))
```

Waiting for remote function results

Waiting for the result of a remote process using the `dwait` keyword generates slightly more complicated code. First, the parameter is copied to a temporary variable since it may be an expression with side-effects and should not be evaluated more than once. Then a call is made to the `_dcx_wait` function to perform the wait operation. The value can then be returned. The generated code uses a statement expression to allow all this code to appear inside an expression.

```
({
    typeof((parameter)) _dct = ((parameter));
    _dcx_wait((struct _dch_var *) _dct);
    _dct->val;
});
```

Waiting for one of many remote processes to finish

The code generated for the `dselect` keyword is quite simple. Each '(variable):(bound)' pair is divided into individual parts and passed as a parameter to the `_dcx_select` function. The first parameter is a count of the number of pairs.

```
_dcx_select((n),
            (variable-1), (bound-1),
            ..
            (variable-n), (bound-n))
```

Distributed For

The distributed for statement uses the 'bucket' handling functions to do most of its work. Once the bucket is created each of the remote functions is started and the location to store the result in saved. All the processes can then be waited for and the results correctly stored. The *Distributed C* code:

```
dfor((exp1); (exp2); (exp3))
    (lvalue) = (function-name)((parameters));
```

generates the following C code:

```
_dct_bucket _dct_b = _dcx_bucket_create();
for ((exp1); (exp2); (exp3))
    _dcx_bucket_add(_dct_b, &((lvalue)),
                    (struct _dch_var *) _dcf_(function-name)((parameters)));
_dcx_bucket_wait(_dct_b);
```

Distributed Block

The code generated for a distributed block is much the same as that for `dfor`. The function `_dcx_bucket_add` is called for each function call along with its l-value.

4.4.7 Lexer

The lexer is automatically generated using *Flex* which has the capability of generating a C++ class. Only one start state is used in addition to the default one: `<COMMENT>` is used to handle comments, this is not the most efficient method for handling comments but it is the simplest. The first part of the lexer handles keywords and passes them onto the parser as objects or objects derived from `Type`, `Qualifiers`, `StorageClass` or `TextToken` as appropriate. Two character symbolic tokens are passed as `TextToken` objects and single character tokens are passed as `CharToken` objects. White space is ignored in the input except for delimiting tokens. String and numeric constants are passed as `TextToken` objects.

Line directives as generated by the preprocessor are handled specially without involving the parser. All lines beginning with a '#' (possibly with leading whitespace) are passed to the `Compiler::HandleDirective` function which extracts the line number and source file for producing meaningful error messages in both the *Distributed C* compiler and the C compiler.

An identifier which does not match a keyword may be either a type name or a simple identifier — the lexer cannot tell. When an identifier is found it is looked up in the symbol table using the global namespace. If the identifier is not found then a simple `TextToken` object is passed, otherwise a `UserType` object is used to represent the type. This relies on the parser to enter new types created using `typedef` correctly in the symbol table.

The only difference between the *Distributed C* lexer and an ANSI C lexer (with *gcc* extensions) is the addition of the new keywords (`dblock`, `dfor`, `dselect`, `dstart`, `dwait`, `process` and `remote`).

The original lexer was written by Jeff Lee (see Section 8.1) but it has undergone many changes to make it suitable for *Distributed C*. The complete lexer definition file, `dc.l`, is in Section B.7.21.

4.4.8 Parser

Writing a parser for ANSI C is not at all straightforward, issues such as `typedef` and 'implicit `int`' make parsing declarations very complex. The final parser parses ANSI C well enough for the differences not to be noticed in practice.

It would take many pages and be very tedious to describe each part of the parser individually. Instead I will describe some of the parts added to support *Distributed C* in detail. The complete parser definition file, `dc.y`, is in Section B.7.20. The *Yacc* rules match closely with the BNF given in Tables 4.2 and 4.3.

The *Distributed C* process variable constructs join to the ANSI C grammar through the `postfix_expression` rule which is allowed to be a `dc_expression` in addition to its other values. The `dc_expression` rule is used to handle the `dstart`, `dwait`, `dselect` and `dkill` keywords. Since each valid `dc_expression` starts with a unique keyword there is no danger of ambiguities with the rest of the grammar. Many of the tokens are redundant once the parsing has been performed and are deleted, for example the `DSTART` tokens and brackets.

The `dc_call` rule is used for the invocation of remote functions inside the `dblock` and `dfor` rules in addition to its use with `dstart`. It creates a single `RemoteStartToken` object (which retrieves its line number information from the identifier). The code generation is performed by the `RemoteStartToken::GetText` function so none is required in the parser.

The `dc_selection_arg` is used to represent a single bounded array (a reference of the form '`a:10`') or a single variable. The code generation is performed by the `BoundedArrayToken::GetText` function. The `dc_selection_list` uses left recursion (which is the most efficient for an LALR(1) parser as generated by *Yacc*) to create a list of `BoundedArrayToken` objects inside a single `BoundedArrayListToken` object. The base case for the recursion is a single `dc_selection_arg` since no arguments is illegal.

The `dstart` construct is handled by part of the `dc_expression` rule. The `DSTART` symbol is required to disambiguate a remote function call from a normal one.[9]

The `dwait` construct requires an arbitrary expression of some process type. The `Compiler::GenerateRemoteWait` function generates the code required to perform the wait operation.

The `dselect` construct uses the `dc_selection_list` rule described earlier to generate a multiple selection. Most of the code generation is actually performed inside the `BoundedArrayToken`

[9]Strictly speaking it would be possible to automatically call all functions earlier declared as '`remote`' remotely without the use of a specific keyword. However, this would disallow the possibility of calling the function locally which may be useful under some circumstances.

and BoundedArrayListToken classes but a small part directly related to the dselect operation is generated by Compiler::GenerateRemoteSelect.

```
dc_call
    : IDENTIFIER '(' argument_expression_list ')'
        { $$ = RemoteStartToken::Create($1, $3); delete $2; delete $4; }
    ;

dc_selection_arg
    : assignment_expression ':'  constant_expression
        { $$ = new BoundedArrayToken($2->GetSourceLine(), $1, $3);
          delete $2; }
    | assignment_expression
        { $$ = new BoundedArrayToken($1->GetSourceLine(), $1, NULL); }
    ;

dc_selection_list
    : dc_selection_arg
        { $$ = BoundedArrayListToken::SwallowBoundedArray($1); }
    | dc_selection_list ',' dc_selection_arg
        { $1->AdoptRight($3); $$ = $1; delete $2; }
    ;

dc_expression
    : DSTART dc_call
        { $$ = $2; delete $1; }
    | DWAIT '(' expression ')'
        { $$ = Compiler::GenerateRemoteWait($1->GetSourceLine(), $3);
          delete $1; delete $2; delete $4; }
    | DSELECT '(' dc_selection_list ')'
        { $$ = Compiler::GenerateRemoteSelect($1->GetSourceLine(), $3);
          delete $1; delete $2; delete $4; }
 /* | DKILL '(' dc_selection_list ')' not supported */
    ;
```

Figure 4.8: *Yacc source for handling process type Distributed C constructs*

Process type variables are attached to the grammar as simple types through the type_specifier_builtin rule (which is also used for struct, union and enum types) which has a production for the symbol process_specifier added. The *Yacc* rule for process_specifier is given in Figure 4.9. A process type is represented by a ProcessType object with the typee it is based on stored as a pointer inside — this information is passed to the constructor, the angle brackets and PROCESS token are no longer required and are deleted to avoid memory leaks.

The 'remote' keyword is treated like other storage class specifiers and is indicated by a bit flag inside the StorageClass object that is created by the lexer. Figure 4.10 contains the rule used for REMOTE.

```
process_specifier
    : PROCESS '<' type_name '>'
        { $$ = new ProcessType($1->GetSourceLine(), $3);
          delete $1; delete $2; delete $4; }
    ;
```

Figure 4.9: *Yacc rule for parsing process type declarations.*

```
storage_class_specifier
    : TYPEDEF { $$ = $1; bInTypedef = true; }
    | EXTERN
    | REMOTE
    | STATIC
    | INLINE
    | AUTO
    | REGISTER
    ;
```

Figure 4.10: *Yacc rule for parsing storage class specifiers, including the* `REMOTE` *token.*

4.4.9 Compiler Rules

Function invocation

A remote function may run on a completely different machine to the caller. This means that the memory space isn't shared between the two, so pointers cannot be passed without copying what they are pointing to. This seems like a simple problem to solve as long as the programmer doesn't cast between types. However, since arrays in C can decay to a pointer to the first element[10] there is no way to tell how large an object the pointer is pointing too. However, an array of fixed size can be passed by wrapping it in a structure, as used in the `struct lstring` example in Section 4.2.

I considered the problem of passing pointers to be too complex in the general case, although I intend to support the passing of strings (variables of type `const char *` or `char *`). This means that the compiler should issue an error message if an attempt is made to pass a pointer.

It is not always obvious whether instances of a particular type may be safe to pass to another machine, it might be a synonym for a pointer declared using `typedef` or it might be a structure containing a pointer element. This can be handled simply by giving types a 'safe' flag and following a few simple rules:

1. A `struct` type is safe if, and only if, all its members are safe.

2. A `union` type is safe if, and only if, all its members are safe.

3. A `typedef` is safe if, and only if, the type it represents is safe.

[10] A 'feature' inherited from B, [14].

If an attempt is made to declare or define a `remote` function that takes a parameter of an unsafe type, or returns one, then the compiler can issue an error message.

Variable declarations

Since different instances of a `remote` function may be running on different machines there is no way that `static` data could be maintained between them. Because of this `static` variables are disallowed inside remote functions. However, since a `remote` function may call other functions in another translation unit it will be impossible to enforce this beyond one function scope.

A possible solution to this would be to only allow functions declared as `remote` to be called from a `remote` function. Unfortunately this would effectively disallow a `remote` function from calling all C library functions. In fact most library functions that use global or static data will do so in a perfectly safe manner. For example, the C `stdio` functions should work perfectly in a remote function provided no attempt is made to export the file pointer.

Variable usage

The runtime environment on each machine will contain a separate copy of all the global variables of the program. This means that if a remote function were to update a global variable the change would not propagate to other functions and the main program. To avoid this situation all accesses to global variables inside a `remote` function are illegal.

A future version of the compiler could provide access to global variables by replacing references to them by special function calls which ensure that all the machines in the distributed system maintain a single value. However, this may not be desirable since it mixes the message passing and shared memory distribution models.

Symbol naming

ANSI C defines all symbols beginning with an underline to be implementation defined and therefore no user program should use such symbols. Since the *Distributed C* system is mid-way between a user program and the compiler implementation I have decided to make all external symbols begin with '`_dc`', this means that any user program using such named symbols may cause problems. Ideally the compiler should warn about this.

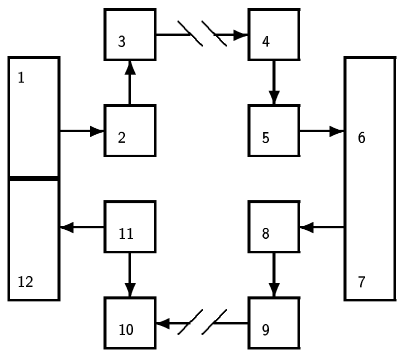
4.5 Runtime System

4.5.1 Overview

The runtime system will be implemented as a single executable program which is capable of looking inside the shared objects generated by the compiler to execute arbitrary functions as required. These functions can then call functions inside the runtime executable effectively using it as a library.

Figure 4.11 shows the stages involved in running a function on a different machine.

The design of the runtime system is deliberately imprecise since its implementation is an ongoing development.



1 Call the function.
2 Package the parameters.
3 Transmit the parameters.
4 Receive the parameters.
5 Unpackage the parameters.
6 Execute the function.
7 Return result.
8 Package result.
9 Transmit result.
10 Receive result.
11 Unpackage result.
12 Continue execution.

Figure 4.11: *Diagram showing the stages of a remote invocation of a function.*

4.5.2 Parameter Passing

The parameters for a remote function will need packaging to be sent to other machines in a way that they can be unpackaged at the other end ready to call the function.

I initially considered creating a stub function which would be called in place of the remote function locally. This function would perform operations on the stack to remove the parameters as a single block of memory which could then be transmitted across the link before being placed in the correct position on the target machine's stack read to call the function. This method was rejected because assembler code would be required and it would need to be unique for each processor and operating system. Also the function would need to be called in the same manner each time which might cause problems for optimisers which might try and make the function inline.

Since the compiler knows the parameters that the remote function requires along with their types it would be possible to create a block of memory of the correct size and to copy the parameters into it. The memory block can then be passed to the remote machine and again because the types are known the parameters unpackaged ready for calling. Rather than perform the memory management myself I decided to let the compiler do it by creating a parameter structure for each function called '`_dcp_<function-name>`' when the function is declared as remote. The parameter structure contains a header of type '`_dch_invoke`' which contains the length of the block so that the actual parameters can be treated opaquely by the transmission code.

An inline package function called '`_dcf_<function-name>`' will be generated for each function which will allocate the necessary space, copy the parameters and then call the necessary runtime functions to start the function remotely.

The return value of a function can conceivably be any 'safe' type (see Section 4.4.9). Because of this every type has a structure named '`_dcv_<mangled-type-name>`' created for it when it is declared which is used for the return value of the inline function.

Once the parameter structure has been received by the remote machine the function needs to be invoked. The actual function is not invoked directly but a function called '`_dcu_<function-name>`' is called instead which unpackages the parameters, calls the actual function and then packages the return value.

Actual declaration

```
struct lstring { char sz[256]; };
remote struct lstring bing(double d);
```

Generated code

```
struct lstring { char sz[256]; };
```

The structure '`_dcv_G7lstring`*' is required for when '*`struct lstring`*' is used as a return value. The name '*`G7lstring`*' is the mangled form of '*`struct lstring`*'.*

```
struct _dcv_G7lstring {
    struct _dch_var hdr;
    struct lstring val;
};
```

The original function declaration without the '`remote`*'.*

```
struct lstring bing(double d);
```

The parameter passing structure along with its header, up to 26×26 *parameters are allowed which exceeds the ANSI C requirement.*

```
struct _dcp_bing {
    struct _dch_invoke hdr;
    double p_aa;
};
```

Inline remote launching function. The return value is a pointer to fit in with the handling of '`process<T>`*' declarations. The local variable* `_dct_p` *is used to point the parameters in the memory allocated using* `_dcx_allocargs`*.* `_dct_r` *is used to hold the return value pointer which is an out-parameter of* `_dcx_exec`*. The actual function name to be executed (which is actually the unpack function) is set using* `_dcx_setfn`*. The large number of casts is required since C does not support subtyping.*

```
static __inline__ struct _dcv_G7lstring (*_dcf_bing(double p_aa)) {
    struct _dcp_bing (*_dct_p) = _dcx_allocargs(sizeof(struct _dcp_bing));
    struct _dcv_G7lstring (*_dct_r) =
        _dcx_allocvar(sizeof(struct _dcv_G7lstring));
    _dct_p->p_aa = p_aa;
    _dcx_setfn((struct _dch_invoke *) _dct_p, "_dcu_bing");
    _dcx_exec((char *) 0, (struct _dch_var *) _dct_r,
              (struct _dch_invoke *) _dct_p);
    return _dct_r;
}
```

Figure 4.12: *The code generated to allow the remote invocation of a function*

Figures 4.12 and 4.13 show the code generated for packaging and unpackaging parameters parameters.

The unpack function is called by the runtime system with the parameter structure, `_dct_pi`*, and return value structure,* `_dct_pv`*. These are then cast to the real parameter and return value structures for the function. The function can then be called with the actual parameters and the return value stored. The return value of zero indicates that the function was called successfully.*

```
int _dcu_bing(struct _dch_invoke (*_dct_pi), struct _dch_var (*_dct_pv))
{
    struct _dcp_bing (*_dct_pp) = (struct _dcp_bing *) _dct_pi;
    struct _dcv_G71string (*_dct_pr) = (struct _dcv_G71string *) _dct_pv;
    _dct_pr->val = bing(_dct_pp->p_aa);
    return 0;
}
```

Figure 4.13: *The code generated to unpack function parameters and retrieve the return value.*

4.5.3 Library Interface

The runtime system provides a number of type declarations and functions which the compiled *Distributed C* program calls. None of these functions are intended to be called by the *Distributed C* program itself, only the generated code.

The structures which allow the creation of arbitrary numbers of threads, then wait for them all to complete are implemented in a slightly different way to the process variable oriented constructs. A 'process bucket' contains any number of processes that have been started along with an address to store the result in when the process is complete. Once all the processes have been started it is possible to wait for all the processes in the 'bucket' to complete before continuing, the results are stored in the designated addresses. This method means that the names of the destination variables do not need to be stored and reconstituted during a 'wait' phase.

Execution

The `_dch_invoke` structure is used as a header for the parameter structures. It contains the 'payload' size, which is the size of the parameters without the header, and the name of the function to be executed. The `_dch_var` structure is used similarly as a header for the return value structure, but it also contains the thread identifier for the thread that is handling the connection, a status indicator, and the position of the process in the completion queue. Much of the time pointers to parameter structures or return value structures are treated as pointers to the `_dch_invoke` or `_dch_var` structures respectively. This allows them to be passed around without their actual type being known, only the size is required when they need to be copied. This is a crude form of polymorphism.

Three of the execution functions map directly to the implementation of the `dstart`, `dwait` and `dselect` keywords.

The `_dcx_exec` function is used inside the packaging function to actually start the function remotely. Its first parameter is the name of the machine to start the function on. There is no language feature that actually allows this parameter to be used but it is honoured if supplied. The compiler generated code always uses *NULL* for this parameter to indicate that the runtime system should choose the target host itself.

The `_dcx_wait` function is used to wait for and retrieve the result from a function that has been started remotely. Its only parameter is the pointer returned from the packaging function.

The `_dcx_setfn` function is used inside the packaging function to set the name of the function that is to be launched inside the `_dch_invoke` structure. The implementation of the function is simply a `strcpy` of the function name into the structure. This could have been put inside the packaging function itself but then a prototype would have been required for the function which could conflict with one the user includes from elsewhere.

The `_dcx_select` function is used to perform the `dselect` operation. It takes a variable number of parameters denoting the bounded arrays (along with size) and simple variables that should be checked. The list is preceded by a count of the number of pairs. Simple variables use zero as the second part of the pair usually used to indicate the size of the array.

Process Buckets

The `_dct_bucket` type is used hold handles which represent a bucket. It is actually a `typedef` for a `void` pointer but this is cast to the correct type internally by the functions — this is to avoid unnecessary declarations in the compiler generated code.

The `_dcx_bucket_create` function takes no parameters, it creates a new process bucket. This function is called at the start of the code generated for a `dblock` or `dfor` construct.

The `_dcx_bucket_add` function takes a bucket handle, result destination address and a process type structure as returned by the packaging function. It stores these values inside the bucket ready for the wait operation to retrieve the result and store it in the correct destination.

The `_dcx_bucket_wait` function actually performs the main actions for buckets. It waits for each of the processes in the bucket (in the order that they were stored) and stores the result in the appropriate location. Once all the processes are complete it frees the memory associated with the bucket and returns.

Memory Allocation Functions

The memory allocation functions `_dcx_alloc`, `_dcx_free` and `_dcx_realloc` are really just wrappers around the corresponding C library functions but with extra logging facilities. They also help to keep the library calls out of the generated program which could cause conflicts.

The `_dcx_allocargs` and `_dcx_allocvar` functions allocate memory for the parameter structure and return value structures respectively but they also fill in the payload field in the header and initialise many of the other members appropriately.

Chapter 5

Implementation

5.1 Libraries

Both the runtime system and the compiler make heavy use of a number of library classes and functions. Some were written specifically for the project but most existed before the programming part of the project started. The following sections attempt to briefly describe the operation of the library facilities but without going into unnecessary detail.

During the design and implementation of the project I discovered that the C++ Standard Template Libraries were working correctly with *gcc*. I found that many of the templated classes it provided were very useful for implementing the compiler. The `ObjectHash` class remained because I required its ability to 'stack' items for representing scope even though the STL provided its own `map` classes. I also continued using the `MString` class even though an alternative was available in the STL because much of the code was already using it.

5.1.1 MString

The `MString` class is a powerful class for representing strings. It is loosely based on the Microsoft `CString` class but shares no code with it. The class provides most of the facilities that would be expected for handling strings: comparison operators, character insertion and deletion, BASIC-like `Left`, `Right` and `Mid` functions, searching and trimming.

The `MString` class is not highly optimised and better performance could probably be gained with greater use of inline functions.

5.1.2 Tracing and Debugging

A major part of the library is enhanced tracing and debugging support. The tracing support is available to both C++ and C code. It uses a number of macros defined in `trace.h` (see Section B.9.6) to allow trace messages to be output to a tracing device (usually just the `stderr` stream). An extension to the tracing facilities allows assertions to be made which will cause a trace error message and the program to abort if they fail.

Memory leak detection is also provided under C++ by overloading the `new` and `delete` operators. A global object maintains allocated memory and warns about any memory left allocated when the program exits. The 'Electric Fence' library[1] is also used for increased

[1] 'Electric Fence' is a library written by Bruce Perens which uses the memory protection facilities provided by UNIX and modern processors to catch attempts to access memory beyond the end of an allocated buffer,

safety.

5.1.3 Alignment

The runtime system function `package_args` (`exports.c`, see Section B.8.3) allocates memory space for a number of structures using `sizeof`. These structures may contain types which have specific alignment requirements.[2] The `ALIGN` macro declared in `align.h` (see Section B.9.10) allows a structure size to be aligned correctly. The default alignment is eight bytes but this can be changed if necessary.

5.1.4 Socket Utilities

A number of small socket utilities written by Andrew Main (see Section 8.1) are used by the runtime system. They repeat attempts to read or write from a socket until the correct amount of data has been sent or received or an error occurs.

5.2 Compiler

5.2.1 Lex & Yacc Grammar

I discovered a version of the C grammar suitable for compiling with *Lex* and *Yacc* at *http://www.lysator.se/*. It was written by Jeff Lee (see Section 8.1). The combined parser suffered from three problems:

- The parser file contained one shift-reduce conflict. This turned out to be the simple dangling `else` ambiguity which is resolved in the ANSI C standard as binding to the closest possible `if`. The ambiguity was removed using the *Yacc* `%right` directive (in fact *Yacc* automatically resolved in this manner without the directive, it just removed a warning message).

- The parser relied on the lexer being able to determine whether an arbitrary identifier was in fact a type name. This situation can only occur when a new type is created using `typedef`. By adding a hash table to the program and using it to store new types as they were defined the lexer could easily make a choice between a type name and a simple identifier.

- The C standard allows a type name to be used as a variable name inside a scope (Figure 5.1). The lexer and parser conflicted with this and a parse error resulted. Despite referring to the *gcc* source code I was unable to solve this problem without introducing ambiguities in the grammar.

---

[13]

[2] I spent many hours trying to find a bug which only surfaced under Solaris which caused the system to 'bus error'. It was caused by my failing to align variables of type `double` on an eight byte boundary. Intel processors do not have this restriction so I had not caught the problem initially.

```
typedef int wibble;

/* wibble is a type here */
void a()
{
    double wibble;
    /* wibble is a variable here */
}

/* wibble is a type here */
```

Figure 5.1: *Example of the ability to reuse the name of a type as an identifier inside a C scope without causing a compiler error.*

Extending the grammar

Before extending the grammar for *Distributed C* I removed an anachronism from the C language to make code generation easier. ANSI C standardised a syntax for function prototypes which made K&R style parameter declarations obsolete. Figure 5.2 compares the syntax of these methods. Since the generation of the parameter structure involves processing argument lists along with their types it was much easier to handle these in the ANSI C form. The removal of K&R style parameter declarations is unlikely to cause any problems.[3]

```
/* ANSI C function declaration */
int main(int argc, char *argv[])
{
    ...
}

/* K&R C function declaration */
int main(argc, argv)
char *argv[];
{
    ...
}
```

Figure 5.2: *Comparison of ANSI C and K&R C function declarations. Note the use of 'implicit int' in the K&R example.*

5.2.2 Parse Tree

In my initial 'translator' a full parse tree was built containing simple `TextToken` objects which consisted of a token identifier (as provided by *Yacc*) and the text of the token. At the end of

[3]In fact the proposed new C standard C9X also removes this feature.

each external declaration (basically any sequence of tokens which terminates back at the top level of the parser) the current tree was output and then discarded.

To make the output close enough to the input I needed to maintain the same line break locations – this would also help debugging. Because the lexer disregarded white-space and comments in the input this information never got as far as the parser. It was relatively simple to add an extra data member to each token containing the whitespace so that it could be prepended on output. Whilst discussing this method I referred to the extra whitespace as 'gubbins' and the name stuck. However, this method was found to be too error prone and was replaced by the source line number in each token.

5.2.3 Compiler Invocation

The actual *Distributed C* compiler forms only part of the complete compilation process. Many combinations of the following stages may be necessary.

1. **Preprocessing**
   The source code is passed through the preprocessor to handle macro expansions, include files, etc. The output from the preprocessor is raw *Distributed C* source with the addition of '`#line`' directives so that the compiler can keep track of line numbers for error reporting.

2. ***Distributed C* Compilation (Translation)**
   The distributed C compiler reads the preprocessed source code and translates all *Distributed C* constructs into C code and function calls.

3. **C Compiling**
   The C code is then compiler by the *gcc* compiler into position independent object code.

4. **Linking**
   The object files for the program are linked together with the various libraries and command line options required.

Since each of these stages is quite complex a driver program analogous to the *cc(1)* program used by the C compiler is used. The driver, `dcc`, is written in PERL and it handles most of the compilation process automatically. It takes a number of the same options as `cc` and performs sensible actions depending on the filenames it is given. The source is in Section B.7.22.

5.2.4 Warnings and Type Checking

I considered the warnings and type checking to be a secondary feature to actually getting the system working. The implementation of these two features is not too complex and could very easily be added to the existing compiler. Possible implementation methods are discussed in Sections 7.1.2 and 7.1.3.

5.2.5 Generating sequential programs

Producing a sequential program from the *Distributed C* source was always one of my intentions. However, it didn't turn out to be as simple as I'd hoped. The `dc2seqc` PERL script (see Section B.7.23 performs some simple alterations to the source to allow it to be compiler as simple C but it does not handle `dselect` fully. The script works by removing the `remote` keyword from all declarations, changing `process` types into their underlying types and removing the `dstart` and `dwait` keywords. The `dselect` function is handled by using a *gcc* statement expression containing a static variable which ranges through the expected responses. The first parameter of the `dselect` statement is used as the upper bound on this range. This method will not work if more than one parameter is given to `dselect`. Section 7.3.7 discusses a better solution.

5.3 Runtime System

5.3.1 Operation

The run-time system is a single executable called `dcd` (*Distributed C* daemon) that can be run in a number of different modes. The most important modes are master and slave. When the daemon is run in master mode it performs initialisation tasks then opens the requested shared object file to execute the `main` function. Any extra parameters not processed by the daemon in `argc`, `argv` and `envp` (as passed to the daemon's `main` function) are used as arguments for the shared object.

Once the actual *Distributed C* program has been started it uses functions within the daemon executable as callbacks to perform the distribution.

Usage

The daemon takes an initial parameter which indicates which mode it should be started in and the name of the shared object it should use. The program also requires a hosts file which is a simple text file containing the name of a host on each line, when the `_dcx_exec` function is used to execute a function remotely it uses this file to decide which machine to execute the function on unless a specific machine name is specified. The shared object name usually ends in '`.so`' and the hosts filename is generated from it by removing the extension and appending '`.hosts`'. It is often useful to make the hosts file a symbolic link to a list of hosts which can be used for many programs. If the host that is to run the master daemon is also to be used as a slave then both a master and slave copy of the daemon must be run. The initial command line option is one of `-m`, `-s`, `-r`, `-p`, `-k`. Two extra command line options, `-P` and `-K` require a host name as the second parameter rather than the shared object filename.

`-m` Starts the daemon in master mode. It will open the shared object file and execute the `main` function with the remaining command line.

`-s` Starts the daemon in slave mode. It will wait for connections of the *Distributed C* port and execute functions as required. This option is not normally used directly, instead `-r` is used to start many remote slaves at once.

`-r` Starts a slave daemon on each of the machines listed in the hosts file using the *rsh(1)* command. The shared object file will be passed to each of them.

`-p` Pings each of the machines in the hosts file to check that they are responding on the correct port and are thus available for executing functions.

- `-k` Kills all the slave daemons running on machines mentioned in the hosts file. This option is useful for cleaning up after running a distributed program.
- `-P` Pings a single host (specified instead of the shared object filename) to see if it is responding to execution requests.
- `-K` Kills the daemon on a single host (specified instead of the shared object filename).

5.3.2 Function Invocation

As described in Section 2.3.2 a shared library can be loaded into the address space of a running process and function addresses can be retrieved from it. These operations are provided by the dynamic library support of the operating system.

The `dlopen` function is used to open a dynamic library (and effectively map into the address space of the processor). Once the library has been opened successfully symbols can be retrieved from it using the `dlsym` function. When the library is no longer required it must be released using the `dlclose` function. Figure 5.3 shows an example program which uses these functions.

Since all *Distributed C* shared objects export unpackaging functions which have a fixed interface the type information isn't required when calling a remote function using this method. The objects also export the `main` function which is used by the master daemon to start the execution.

5.3.3 Symbol Naming

Since the daemon and the compiled *Distributed C* program share a single namespace for external symbols the daemon's symbols must be named to avoid clashes. As described in Section 4.4.9 all the runtime system's externally visible symbols start with the sequence '`_dc`'. This sequence is usually followed by a third letter and another underline before the actual name of the symbol. The third letter determines the type of the symbol. Note that this is not enforced but it makes some parts of the code easier to understand. Valid sequences are described in Figure 5.4.

5.3.4 Concurrent Execution

The first version of the runtime system implemented the parallelism by executing threads concurrently on a single machine. This method is still available if the system is compiled with the preprocessor symbol '`CONCURRENT`' defined. The only appreciable difference between the 'concurrent' and 'distributed' modes is in the `thread_start` function in `exports.c` which starts the function directly using `_dci_startfn` if running concurrently, or causes the request to be sent to another machine using `_dcn_send` if running in a distributed manner.

The best way of describing the stages involved in executing functions in parallel is to follow through an example program. The source code for this program is given in Figure 5.5 with an edited version of the compiler generated code in Figure 5.6 (various unused declarations have been removed along with the function prototypes, the full header can be seen in Section B.7.2).

At the beginning of the `main` function two variables are declared. The first variable is the compiler generated form for '`process<struct lstring>`'. It is used to store the return value from the packaging function, which in turn calls `_dcx_exec`. The `_dcx_exec` function creates

```c
#include <dlfcn.h>
#include <stdio.h>
#include <time.h>

/* The types of the functions we are calling */
typedef void (*PFN1)(unsigned int);
typedef int (*PFN2)(void);

int main()
{
    /* Handle is used to refer to the library */
    void *handle;
    PFN1 fn1;
    PFN2 fn2;

    /* Open libc (change for Solaris), RTLD_LAZY means that the functions */
    /* should be relocated only when required.                            */
    handle = dlopen("/lib/libc.so.5", RTLD_LAZY);

    /* Check to see if we found the library OK */
    if (!handle)
    {
        printf("%s\n", dlerror());
        return 2;
    }

    /* Retrieve the addresses of the functions */
    fn1 = dlsym(handle, "srand");
    fn2 = dlsym(handle, "rand");

    /* Seed the random number generator */
    fn1(time(NULL));

    /* Output a couple of random numbers */
    printf("%d\n", fn2());
    printf("%d\n", fn2());

    dlclose(handle);

    return 0;
}
```

Figure 5.3: *Example of the use of the dynamic library access functions. The program retrieves the addresses of the* `rand` *and* `srand` *functions from the C library (which should already be loaded) and uses them.*

---

a structure which is filled with the information necessary to start the function before being passed to the POSIX thread creation function. If successful the `_dcx_exec` function returns allowing the program to continue.

dcf The name of an inline packaging function generated by the compiler. The name of the underlying function follows the prefix.

dci Internal functions and types. e.g. _dci_check_status and _dci_thread_args.

dch Header structures which are intended for use as the first element in another structure. e.g. _dch_invoke and _dch_var.

dcn Networking functions which directly send or receive data. e.g. _dcn_send and _dcn_pingn.

dcp The parameter structure for a remote function generated by the compiler when it encounters a remote function declaration. The name of the underlying function follows the prefix.

dct Temporary variables used inside the packaging and unpackaging functions.

dcu The name of the unpackaging function generated by the compiler when it encounters a remote function definition. The name of the underlying function follows the prefix.

dcv The name of the return value structure generated by the compiler when it encounters a complete type declaration. Such declarations for the built in types are contained in the preamble file. The mangled version of the underlying type follows the prefix.

dcx External functions intended to be called by the compiler generated code. e.g. _dcx_exec and _dcx_select.

Figure 5.4: *Prefix sequences used for external symbols in the runtime system and symbols automatically generated by the compiler.*

The new thread begins executing the static thread_start function which receives the information structure as a void pointer which is cast to the correct type. If the runtime system is compiled for concurrent processes the _dci_startfn function is used to start the remote function. However, if the system is compiled to start processes remotely then the _dcn_send function is used to pass the request to another machine. When the appropriate function returns a mutex and condition variable are used to signal to waiting parties that the thread has completed. This is explained in Section 5.3.5.

The compiler turns the use of dwait into a number of statements inside an expression.[4] A new variable is declared to hold the process variable parameter given to dwait, this is done to avoid side-effects.[5] The _dcx_wait function is called to wait for the remote function to

[4]Being able to place arbitrary statements inside an expression is a *gcc* extension to the C language. It is very useful in this case since dwait may be used anywhere in an expression and the process type that it uses may have side effects. For example it would be difficult to generate code that correctly handles printf("%d", dwait(++p));. The last expression in the compound statement is used as the value for the expression.

[5]The typeof operator is another extension to C by *gcc*, it performs a similar action to the standard sizeof

```
#include <stdio.h>
struct lstring
{
    char sz[256];
};

remote struct lstring bing(double d);

int main()
{
    process<struct lstring> a;
    struct lstring i;

    a = dstart bing(3.141);
    i = dwait(a);

    printf("%s\n", i.sz);

    return 0;
}

remote struct lstring bing(double d)
{
    struct lstring a;
    sprintf(a.sz, "%f, %f\n", d, 2.0*d);
    return a;
}
```

Figure 5.5: *Source for the concurrent execution example program.*

---

complete and store the result correctly inside the structure. The final part of the statement expression retrieves the actual return value of the remote function.

5.3.5 Multiple Waiting

The implementation of `dselect` required a method of waiting for one of many threads to complete. Unfortunately this capability is missing from the POSIX threads standard.[6] To circumvent this I needed to use a combination of mutexes, condition variables and broadcasts to achieve the same functionality.

Figure 5.7 shows the code that was necessary to implement multiple waiting. It was important to ensure that many copies of `_dcx_select` could be running at the same time as well as many threads completing at the same time. It was also important that a thread could not complete between checking its status and executing the `pthread_cond_wait` function and result in deadlock.

5.3.6 Distributed Execution

The additions to the concurrently executing daemon to support remote execution turned out to be far simpler than I originally expected. The additional code fell into two parts: socket

operator except it yields the type of an object.

[6]Should the system be ported to Windows NT the `WaitForMultipleObjects` system function is provided which can be used in this manner.

```
struct lstring {
    char sz[256];
};

struct _dcv_G71string {
    struct _dch_var hdr;
    struct lstring val;
};

/*remote */ struct lstring bing(double d);
struct _dcp_bing {
    struct _dch_invoke hdr;
    double p_aa;
};

static __inline__ struct _dcv_G71string (*_dcf_bing(double p_aa)) {
    struct _dcp_bing (*_dct_p) = _dcx_allocargs(sizeof(struct _dcp_bing));
    struct _dcv_G71string (*_dct_r) =
        _dcx_allocvar(sizeof(struct _dcv_G71string));
    _dct_p->p_aa = p_aa;
    _dcx_setfn((struct _dch_invoke *) _dct_p, "_dcu_bing");
    _dcx_exec((char *) 0, (struct _dch_var *) _dct_r,
              (struct _dch_invoke *) _dct_p);
    return _dct_r;
}

int main()
{
    struct _dcv_G71string *(a);
    struct lstring i;

    a = _dcf_bing(3.141);
    i = ({
     typeof(a) _dct = (a);
     _dcx_wait((struct _dch_var *) _dct);
     _dct->val;
     });
    printf("%s\n", i.sz);

    return 0;
}

/*remote */ struct lstring bing(double d)
{
    struct lstring a;
    sprintf(a.sz, "%f, %f\n", d, 2.0 * d);
    return a;
}

int _dcu_bing(struct _dch_invoke (*_dct_pi), struct _dch_var (*_dct_pv))
{
    struct _dcp_bing (*_dct_pp) = (struct _dcp_bing *) _dct_pi;
    struct _dcv_G71string (*_dct_pr) = (struct _dcv_G71string *) _dct_pv;
    _dct_pr->val = bing(_dct_pp->p_aa);
    return 0;
}
```

Figure 5.6: *Output generated by compiling Figure 5.5.*

---

code which handled the actual communications (`dist.c`, see Section B.8.5); and hosts code which managed the lists of hosts (`hosts.c`, see Section B.8.4).

There is a single copy of this thread status structure. It is used to communicate between threads and functions waiting for threads.

```
struct _dci_thread_status {
    pthread_mutex_t lock;
    pthread_cond_t finished;
    pthread_t tid;
    unsigned int qcurrent;
};
```

When a thread has finished doing useful work it executes the following sequence which ensures that all interested parties know that the thread has completed but will not block if no-one is. The first lock statement will block if any other thread is attempting to broadcast or if a `_dcx_select` *function is currently checking its thread set, both of these operations eventually terminate.*

```
pthread_mutex_lock(&thread_status.lock);
pthread_cond_broadcast(&thread_status.finished);
pthread_mutex_unlock(&thread_status.lock);
```

When the `_dcx_select` *function wishes to wait for one of a set of threads to finish it waits for broadcasts on the condition variable and then checks. This method avoids busy waiting. Note that the* `pthread_cond_wait` *function unlocks the mutex while it is waiting and then atomically locks it when the broadcast is received before returning.*

```
pthread_mutex_lock(&thread_status.lock);
while (All threads we're interested in haven't finished)
{
    pthread_cond_wait(&thread_status.finished, &thread_status.lock);
}
pthread_mutex_unlock(&thread_status.lock);
```

Figure 5.7: *Description of the mutex and condition variable methods used to perform multiple waiting.*

When the runtime system is only compiled for concurrent use the `_dci_startfn` function is called inside `thread_start`. When the system compiled for a distributed system `_dcn_send` is called in its place which causes the request, along with its parameters, to be sent to another machine for execution.

The `_dcn_send` function first attempts to make a connection with the machine specified in its parameters, this will fail if a machine hasn't been specified or the machine is refusing connections. If no connection can be established then the `_dcn_gethost` function is called to retrieve a host from the host list. An attempt is then made to contact the new host, if this fails too then `_dcn_gethost` is called again to retrieve a new host and the process continues. If none of the hosts are responding then the system can get into an infinite loop, however this is an indication that there is something very wrong.

Once the outgoing connection has been made the communication begins. The protocol is described in Figure 5.8. If either party fails to comply with the protocol then the connection is terminated. When `_dcn_send` returns the function has been called and the contents of the result structure are filled in just the same way as if the function had been called locally.

Local
DCPREAMBLE Size of invoke structure Contents of invoke structure Size of the result structure Contents of result structure header DCPOSTAMBLE
Remote
Function called DCPREAMBLE Status variable Size of result structure Contents of result structure DCPOSTAMBLE

Figure 5.8: *Description of the socket protocol the Distributed C system uses to execute a function remotely. The* `DCPREAMBLE` *and* `DCPOSTAMBLE` *parts are four byte integers which are used to ensure that both parts are talking the correct protocol.* `DCPREAMBLE` *has the value* `0x00DCDC00` *and* `DCPOSTAMBLE` *has the value* `0xDC0000CD`.

When the runtime system is started in slave mode it runs the `_dcn_receive` function which binds to the socket and waits for incoming connections. When a new connection is received a new thread is created to handle the connection and the main thread continues to wait for new connections. The new thread is passed the file descriptor of the open socket and it attempts to read a four byte integer from it. The integer will contain `DCPREAMBLE` if an attempt is being made to start a function and this type of connection is handled in accordance with the protocol in Figure 5.8. However, other values are also permitted. Receiving `DCPING` (`0x01DCDC10`) causes a simple response of sending `DCPONG` (`0xDC0110DC`) and then closing the connection. This is the method used when the command '`dcd -p`' is issued to detect if the host is accepting connections. If `DCKILL` (`0xDCDEADDC`) a `DCPONG` response is sent and the daemon kills itself immediately.

5.4 Example Programs

During the implementation a number of very simple test programs were written to test both the compiler and the runtime system. Also, a number of example programs were written for the purpose of demonstrating the program.

5.4.1 Mandelbrot Example

This program generates a Mandelbrot set picture using the *Distributed C* system. It splits the screen area into a number of equally sized chunks which are each calculated by a separate remote function. The results are then displayed as they are received by the main program.

Figure 5.9 shows the program running, the empty boxes are chunks which are still being processed.

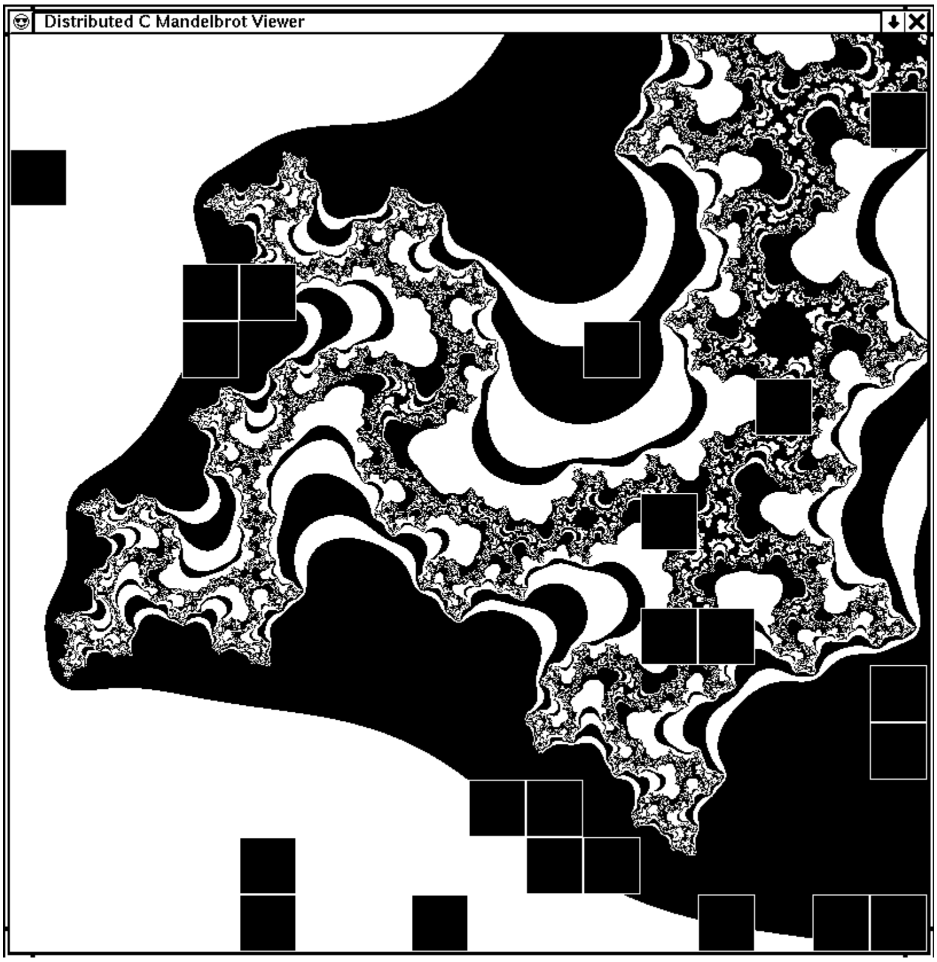


Figure 5.9: *The Mandelbrot program in action. The empty boxes are chunks that are still being processed. Unfortunately it is not possible to show the program in its full colour.*

The program uses the X Windows System to open a window on the screen and display the chunks and resulting picture. The X code is in a separate file to the main source code for two reasons: at the time the example program was started there was a problem with the compiler in relation to incomplete types which revealed itself on the X header files; also, the compilation speed is greatly increased by not requiring the complete X header files (about 8000 lines) to be passed through the *Distributed C* compiler.

So that timing tests could be performed the program was written in a manner to allow compilation as both *Distributed C* and straight C. This was done using `#ifdef` blocks and not using the `dc2seqc` script.

5.4.2 Addition Example

This example program was written as the first program to be seen by people new to *Distributed C*. It shows how to declare remote functions, call them, see which ones have finished and retrieve the results. There is very little other code to get in the way of the new features.

5.4.3 Factorial Example

This program was written to check that functions running on remote machines could correctly start other remote functions on other machines and to show that this was possible. Of course, the program is actually useless since the overhead of starting remote functions is far too high for such a simple problem.

5.5 Overhead Example

This program was written to try and quantify the overhead in time of executing a function remotely. It is very simple and may be compiled as both *Distributed C* and C.

Chapter 6

Testing

I attempted to implement small parts of each of the components at a time so that they could easily be tested. Both the compiler and runtime system were developed in tandem, with features usually being added to the runtime system and then the necessary constructs added to the compiler.

6.1 Library

Each of the library classes is self contained and is therefore relatively easy to test. A stand-alone test program was written for each of the library classes during their development. The `ObjectHash` class was most important to the compiler. Its complexity meant that it needed heavy testing and this is done by the `test_objhash` program.

The `MString` class had already been used in a number of projects and was considered reliable. Even so, I wrote a test program for the new features I added to the class for this project.

The other library classes were all tested as they were written.

6.2 Compiler

The compiler is the most complex part of the system. Most of the incoming code is passed straight through, so once a few large C programs had been correctly handled I was happy that this part was working correctly. It was the type parsing system and the code to handle the *Distributed C* features that was most likely to contain problems.

The type parsing system was designed as a whole after long discussions with Andrew Main (see Section 8.1) about the intricacies and idiosyncrasies of the C type system. The code was tested by checking that types passed into the system were equivalent to those passed out. A tool called `cdecl`[1] proved very useful during this process.

The actual code generation for the *Distributed C* structures was implemented and tested gradually. Figure 6.1 shows a source file that was used for testing when only part of the code generation was complete. In this case code was correctly generated for the `remote`, `process` and result structures but the `dstart` and `dwait` keywords were unimplemented. So

[1]A simple program which converts C types to plain English and vice-versa.

that the program could be tested with the runtime system (which at the time only implemented concurrent execution, not distributed) the code that should have been generated was substituted.

```
#include <stdio.h>
struct lstring
{
    char sz[256];
};

remote struct lstring bing(double d);

int main()
{
        process<struct lstring> a;
        struct lstring i;
#if 0
        a = dstart bing(3.141);
        i = dwait(a);
#else
        a = _dcf_bing(3.141);
        i = (_dcx_wait((struct _dch_var *)&a), a.val);
#endif
        printf("%s\n", i.sz);

        return 0;
}

remote struct lstring bing(double d)
{
    struct lstring a;
    sprintf(a.sz, "%f, %f\n", d, 2.0*d);
    return a;
}
```

Figure 6.1: *A source file used part way through the implementation of the code generation system. When the correct keywords are implemented the* '`#if 0`' *line can be changed to* '`#if 1`'*.*

6.3 Runtime System

The runtime system was implemented in two stages as described in Section 5.3.4. This meant that the basic system (including the compiler) could be tested and proved to be sound before embarking on the more complex distributed system. Testing of the concurrent runtime system involved writing lots of test programs which started many remote functions (which were of course run locally) and waited for them to finish.

Once the basic system proved reliable I implemented the distributed part and debugged that. The ultimate test of the complete system was the mandelbrot viewing program (see

Section 5.4.1). However, the program seemed to display the chunks in a simple raster order instead of the erratic order I was expecting since the simpler parts of the picture should have been calculated more quickly. I realised that this was probably because the implementation of `dselect` simply found the *first* process that had completed in its argument. The seemingly considerable time it took to draw the chunks on the screen, coupled with the fact that many of the processes completed at similar times meant that the drawing lagged behind the available results. Although this behaviour would not be a problem in most applications, I found it difficult to persuade people that the chunks were really being calculated at different rates using a number of machines. To make it more obvious I created a queue of finished processes so that `dselect` would return the earliest finishing process that had not already been handled rather than the first one in the array. This method made the concurrent nature more obvious. I also made various improvements to the code that used the X Window System drawing facilities.

6.3.1 Performance

The performance of the system was not a major factor in the design, but I did try to keep the amount of data actually sent across the wire to other machines to a sensible minimum since I believed this to be where the major bottleneck would lie.

There are two sides to the performance of the system: time lost by executing the function remotely; and time gained by having the two functions executing concurrently. I attempted to measure both of these using the test programs.

To measure the overhead of the runtime system in executing a function I used the example program in Section 5.5. The source code can be compiled as both C and *Distributed C* making the comparison possible. The actual function that is to be tested simply sleeps for a few fractions of a second and then returns. The tests were performed with a delay of fifty milliseconds and 200 iterations — a total delay of ten seconds. As would be expected the C version had negligible overhead and took only eleven seconds to execute. However, the *Distributed C* version took a disappointing 33 seconds to complete. This means that a remote function invocation has an overhead of around 100 milliseconds. The timings were consistent when the slave daemon was run remotely or on the same machine.

To measure the benefit of performing calculations across many machines I used the mandelbrot viewer program from Section 5.4.1. The results are given in Figure 6.2. The times achieved when using a number of machines are encouraging. The base case of only one machine is disappointing but the large amount of time is probably due to the amount of context switching being performed and the thread creation overhead. In fact the one minute load average of the machine exceeded nine when the program was executing in this configuration. The fact that two dual processor machines were roughly equivalent to four single processor machines is to be expected. But, this might indicate that the overhead is in the communication rather than in the calculation since the dual processor machines were actually far more powerful than the single processor machines. However, I am most pleased by the result for seventeen machines. Section 7.3.1 describes some methods which may improve the performance of the system.

Configuration	**Time (s)**
Sequentially on one processor (compiled as C)	30
Distributed runtime system on one host running as both master and slave	186
Distributed runtime system using a separate server machine as the slave (gem).	19
Distributed runtime system using two server machines as slaves (gem and stone).	19
Distributed runtime system using four workstation machines as slaves (all SS2).	20
Distributed runtime system using seventeen machines (gem, stone and SS2 workstations).	8

Figure 6.2: *The results of timing the mandelbrot viewing program with various configurations. Each configuration was tested six times and the mean is given. In all cases the master host machine was a Sun SparcStation 2 (40MHz SPARC). Gem is an Enterprise 3000 server (2×168MHz UltraSPARC) and Stone is a SparcStation 10 (2×50MHz SuperSPARC).*

Chapter 7

Evaluation

7.1 Work completed

The system, as originally designed, works almost in its entirety. The main omissions are the support for the `dkill` keyword and proper type checking and error messages. There are also two minor problems with the compiler.

7.1.1 dkill

The `dkill` keyword was intended to be able to kill off remote functions when their results are no longer required. This feature is quite difficult to implement but I had a number of ideas. The main problem is receiving the request to terminate the function at the remote host. At the time that the function is executing nothing is reading from the socket. One solution involves starting another thread to actually execute the function whilst the accepting thread continues to monitor the socket. If a terminate request is received it is then very easy to terminate both threads. Unfortunately this method adds even more threads to the system, although one will spend most of its time blocking inside the kernel and therefore not affect performance.

7.1.2 Compiler Warnings

It would be very useful if the intended compiler warnings were implemented. The main warning would be if an attempt was made to define a remote function which accepted pointers as arguments or as a return type. Also, a variable argument remote function is likely to cause errors in the C compiler rather than in the *Distributed C* compiler. The method of marking types as being 'safe' or 'unsafe' as described in Section 4.4.9 could be implemented relatively simply with the current compiler design.

7.1.3 Type Checking

It is currently very easy to introduce bugs into a *Distributed C* program when using `dselect` since little type checking is performed. It is difficult to inadvertently cause problems with the `dwait` construct since the generated code will fail to be compiled by the C compiler if the parameter is not of the correct type — the error message is not particularly helpful, however. The process variable structures are effectively subtypes of the `_dch_invoke` structure, however, the C compiler does not know this and the actual type is lost when they are passed through the `_dcx_select` variable argument list. This method means that if a slightly incorrect type is used the error will not be caught until runtime when the program will probably cause a segmentation fault, bus error or worse, continue executing with incorrect results. Adding sufficient type checking adds a certain amount of complexity to the compiler and symbol table but is not impossible. Unfortunately, it may be necessary to support the complex type promotion rules that C uses if full type checking is performed.

A possible shortcut would be to store the type of every process variable in the symbol table. If `dselect` is used on a variable not in the symbol table then an error can be produced, if it is in the symbol table then the type is known. This method has the major benefit of being simple to implement and requires no new data structures, it could also be used to check the parameter for `dwait` so that a better error message could be produced by the *Distributed C* compiler rather than leaving it to the C compiler.

7.1.4 Compiler Problems

Incomplete types

When a source file is converted from *Distributed C* to C by the compiler every type has a 'result structure' generated for it so that it can be used as the return type for a remote function (see Section 4.5.2). Since this structure contains an instance of the type being declared it cannot be performed on incomplete types. An incomplete type is the result of a forward declaration of a type, for example '`struct incomplete;`' declares `incomplete` to be a structure type name. An incomplete type can be used to declare pointers and function prototypes but it cannot be instantiated since the size is unknown. This rule means that a remote function cannot be declared to return an type which is incomplete at that point in the source code. In reality this is unlikely to happen very often. The only way round the problem would be to defer the generation of the result structure and the generation of the packaging function until after the type has been completely declared.

Type redefinition

This problem with the parser has already been described in Section 5.2.1. It should be possible to get round this problem with a bit of work on the parser.

7.2 Comparison with other languages/systems

7.2.1 Concurrent C

Distributed C is actually remarkably similar to Concurrent C. Both are implemented as preprocessors which output Cand treat a process much like a function with extra properties. Of course, Concurrent C is a far more complex language than *Distributed C* but it does not directly support distributed programming.

The Concurrent C feature of 'transactions' which create entry points within a process and allow them to be called synchronously is a possible way forward for *Distributed C*. This would allow a distributed process to act as an ongoing 'server' of information rather than the current method of it only having a return value.

7.2.2 PVM/MPI

Although the *Distributed C* language was not designed to compete with PVM or MPI it is useful to compare the facilities offered. PVM and *Distributed C* share the ability to work using a user-configurable pool of hosts, although there is no direct support for altering the pool during execution in *Distributed C*. The two systems also share the ability to take advantage of multiprocessor machines whilst still allowing a single processor to execute more than one task.

The *Distributed C* support for implicit message passing means that the programmer only has control of the sending of messages (calling remote functions), there is no way for the programmer to explicitly wait for a message to be received. Such facilities are the basis of PVM and MPI. *Distributed C* and MPI share the restriction that all hosts must be common architectures, but since *Distributed C* does not make use of any special networking hardware only data representation problems (endianness, floating point numbers) must be circumvented to overcome this. A useful feature of PVM is that it can execute tasks on the most appropriate hardware (for example, matrix operations on a SIMD (Single Instruction, Multiple Data) machine), *Distributed C* has no such support.

Neither PVM nor MPI seem to have any load balancing support of the type explained in Section 7.3.1.

Whilst PVM and MPI are far more powerful than *Distributed C*, it is much easier to write *simple* distributed applications and run them using *Distributed C*.

7.2.3 ParC

ParC is a rather different language to *Distributed C*. It is a concurrent language that provides no support for distributed programming and its design does not lend itself to use on a machine that does not provide shared memory parallelism.

The *ParC* language is much more powerful and can be used to express more complex parallel structures than *Distributed C* but to do this it lacks the simplicity of *Distributed C*.

The programmer seems to be able to tune the program to the host machine. For example, it is possible to discover the number of processors so that concurrent tasks can make best use of them. In the design of *Distributed C* I felt that to make the language simple enough such information should only be available internally to the runtime system.

7.3 Desirable Features

7.3.1 Performance

Thread overhead

Thread creation is a reasonably time consuming operation since the kernel must create the data structures required to hold the context information. However, this is unavoidable if threads are being used. It may be possible to reduce the thread creation and destruction overhead by starting a number of threads and then re-using them — this method could fit in well with the 'Load limiting' described below.

In the master daemon the threads spend most of their time sat idle waiting for a response from the remote machine. This means that they are blocked in kernel space so they should never actually be switched to. However, when the remote function completes the thread becomes runnable again and continues its execution. When many functions finish at the same time all the threads are executing for a short amount of time which may lead to unnecessary context switching. It may prove more efficient to serialise some of the operations using critical regions to avoid unnecessary context switching.

Copying overhead

Figure 7.1 contains some code generated to implement the `dwait` keyword. At the end of the expression statement the C compiler effectively generates the code for: '`i = _dct->val`'. If the return value of the remote function is a large structure type (which is quite likely since pointers aren't allowed) then the value must be copied. This is probably unavoidable since it is difficult to know what will be done with the return value of the function. Hopefully a careful optimiser could remove the copy in most cases (although *gcc* doesn't seem to, even with '`-O2`' optimisation).

The function `_dcx_bucket_wait` is used to wait for remote functions to complete and copy the results in the `dfor` and `dblock`. It is also performing a redundant copy operation, but it may be possible to remove it. It is possible that an extra parameter could be added to the `_dcx_exec` function which would optionally contain an address at which to store the result directly. This information is available at the point of execution and would require little extra work.

```
/* Original code:  i = dwait(a); */
i = ({
    typeof(a) _dct = (a);
    _dcx_wait((struct _dch_var *) _dct);
    _dct->val;
    });
```

Figure 7.1: *The C code generated to handle the Distributed C* `dwait` *keyword*

Load balancing

It is likely that the slave machines in the distributed system may vary greatly in processing power and other load. The current system just starts processes on the next machine in its list whatever the status of the machine (unless it fails to respond, in which case it is skipped). I tried to make the more powerful machines do more of the work by putting their host name in the hosts file more than once which improves their utilisation.

Attempting to start processes on a highly loaded machine could create a large bottleneck. Each machine maintains its own indication of its load through the one minute load average which can be queried.[1] I have considered two ways of balancing the load:

1. Whenever a remote process is to be started a broadcast message is sent using UDP (User Datagram Protocol) which will be received by all slave daemons. Each daemon can then respond with an indication of its load — a suitable indication might be the

[1] On Linux this information is freely available through the `/proc` file system but under some operating system it must be retrieved from kernel data structures which requires a setuid root program (e.g. `uptime`).

load average divided by the number of processors, possibly with an extra factor for the processing power of the machine. The function can then be executed on the most appropriate host for the conditions.

2. A separate thread can be run by the master daemon which monitors the load average of each of the slave machines using an RPC call to the `rpc.rstatd` daemon. Any daemon (master or slave) can then query this thread to identify a suitable machine for executing the function on.[2]

Of course neither of these methods will give the optimum distribution strategy since the load on a machine is dynamic and can only be estimated based on its previous values. For example, a remote function that takes a few minutes to execute may be started on a virtually idle machine at the same time as another user begins a large compilation. Also the machine may be being fully utilised by a low priority (high 'nice' value) process which will immediately defer when a higher priority task is started.

Load limiting

The current system starts as many threads on a host as the program requests. Since there is no way for the program to determine how many hosts are available,[3] it is possible that many threads may be running on a single host greatly exceeding its processing capacity. This situation leads to many context switches and less time is spent actually executing useful code.

As the designers of *ParC* noted with the `lparfor` loop, it is more efficient to allow only a certain number of concurrent processes to execute, serialising the total number to achieve this. It would be possible for each slave to only execute one thread at a time either by only providing one thread and causing the socket I/O to block or through creating a new thread each time and using a semaphore. In fact, a counting semaphore could be used so that multiprocessor machines could have one thread running for each processor.

7.3.2 Multiple Use

The current implementation stops the system from being used by more than one program sharing the same machines. My original design allowed the slave daemons to open any shared object and execute the functions within. This would mean that the slave daemons could run continuously on many machines and service requests from arbitrary programs that required remote functions to be executed. This would have been implemented by passing the name of the shared object file as well as the name of the function to across the socket connection. There were two problems with this approach, security and versioning. The security implications of being able to execute arbitrary code as a user would obviously be important in the final system but they were not of particular importance during development. The versioning problem was far more serious. I was worried that if a new version of the program was compiled then the master daemon may be running the new version whilst the slaves were running the old version. This was not a problem on a single machine as long as the handle to the shared object was

[2]If the response is for the same machine it should even be possible to avoid all the socket communications and simply use the concurrent calling method.

[3]This was a design decision, I believed it to be the runtime system's job to share the load rather than the programmer's

closed each time a function was used. But I am not convinced that the problem wouldn't occur when using NFS mounted file systems. This area would need further investigation.

Use by different users would either require the daemon to run with superuser privileges (which would be very dangerous) and then switch to the correct user ID to execute the code or each user to use a different port number.

7.3.3 Memory leaks

In the current system the memory allocated with `_dcx_allocvar` during remote function execution is never actually released when the result is read. This means that if a program runs for a very long time and continually calls remote functions it will gradually work its way through memory causing swapping. For the short demonstration programs this is not really a problem since all memory is automatically released when the program exits.

It is difficult to find somewhere to delete the memory without adding some form of garbage collection or by adding redundant copy operations. The memory should be freed during the code generated for the `dwait` keyword but the return value has not been copied from the memory until after this code has finished executing. A `dwait` expression may be used as a function parameter or part of a much larger expression so simply deferring the deletion may not be enough.

A better solution may prove possible with the better type handling as described in Section 7.1.3.

7.3.4 Thread cleanup

When the runtime system is running as a slave it starts a new thread for each incoming connection. However, it never actually performs a 'join' operation to retrieve the return value from the thread. This means that every thread that is created during the execution remains, although it won't actually use any processor time since it will be blocked waiting for another thread to 'join' with it. There is a maximum number of threads which may exist at any one time so this may cause a problem with large programs.

This situation could be solved by a similar means to that used in the `_dcx_select` function. When each thread is about to finish it performs some semaphore operations and sets its thread ID in a global variable. A separate thread, created especially for disposing of completed threads, can then perform the 'join' operation on the thread thus freeing up the resources it is using. This method is very simple and would require little effort to implement.

7.3.5 Compiler

Keywords

One of the major aims of language designers is to avoid the introduction of new keywords. New keywords mean a loss of compatibility with every program that happens to use the keyword as an identifier or type name. I realised after the implementation of the language that fewer keywords could be used without causing ambiguity in the language.

A single keyword, `dist`, could be used as a replacement for the `dblock` and `dstart` keywords. It could also be used as a prefix for the `for` keyword instead of `dfor`. The `dwait` and `dselect` keywords would need to be retained but renaming `dwait` to `dresult` might better indicate its use. Figure 7.2 shows the new keywords and their equivalent old keywords.

Original	**New**
`dblock { ... }`	`dist { ... }`
`dfor(...) ...`	`dist for(...) ...`
`dstart fn(...)`	`dist fn(...)`
`dselect(...)`	`dselect(...)`
`dwait(expr)`	`dresult(expr)`

Figure 7.2: *Possible more minimalist keywords and the originals.*

Executing non-functions

It is sometimes tedious and long winded to write a separate function declaration for code that will be executed remotely, particularly if the code is short. Languages like *ParC* allow arbitrary code to be executed remotely and automatically generate the appropriate functions. However, for I don't think this is really appropriate for *Distributed C* since any code which is of sufficient complexity to warrant its execution remotely should be placed in a separate function — little code that can be expressed in a few lines is unlikely to yield any performance benefit when executed remotely.

Terminology

It has become clear to be that the terminology used to describe the actions of the compiler and runtime system could be clearer. Words like 'concurrent', 'distributed', 'remote', 'process' and 'function' need to be tied to specific meanings and maybe some of the removed.

7.3.6 Performance

The performance of the compiler is reasonable (compiling a simple program that includes many large system header files takes about five seconds) but I think it could be improved. Performance was not a major issue in the design of the compiler, a clean design and working program were of more importance. But not the compiler works correctly more use of '`inline`' functions could probably yield great performance benefits — unfortunately, by their nature, virtual functions cannot be inline. Also much of the code generation is performed by copying existing parts of the parse tree which then get deleted. Closer analysis of these actions could remove redundant copying. Also deferred copying (i.e. copy-on-write) in tandem with reference counting may reduce redundant copying. A small improvement may be achieved by using the C++ STL string class in place of the `MString` class.

The lexical analysis creation tool *Flex* is known to produce a good finite state machine for tokenising the input. It may improve performance and storage space to remove the keywords from the lexer and tokenise them as simple identifiers which are then checked for in a lookup table of keywords which are then passed to the parser. I was informed of this method by Andrew Main who has used it in his own compiler for the K language.

7.3.7 Generating Sequential Code

The `dc2seqc` script is not a complete method of converting *Distributed C* source code to sequential C. It does not handle `dselect` with more than one parameter and may get confused by perfectly valid code with strange spacing, line breaks or brackets. The correct way to generate sequential C would be to provide an option to the compiler which would output it instead of the code to support distributed use. This would be relatively easy but would require adding conditions to large parts of the parsing code. However, it would work in all cases and keep line number information from the original file.

7.3.8 Porting

I did not manage to port the system beyond Linux and Solaris. Possible platforms included IRIX and Windows NT.

Porting the compiler to another variant of UNIX is likely to just require a recompile of the source code provided a suitable C++compiler is available. The compiler source code does not use any platform specific features so even porting to Windows NT shouldn't pose too much of a problem. However, a 16 bit operating system may not be suitable due to the large amount of memory used.

Porting the runtime system is not as straightforward. If the platform provides standard support for the technologies used (POSIX threads, dynamic libraries, sockets, etc.) then porting should be relatively easy. The *gcc* specific features will require *gcc* of course, but it is available for every UNIX I have come across. However, as far as I know Windows NT does not provide such support and the code will need altering to use that natively provided. There is a project known as 'GNU-Win32' which provides *gcc* and other UNIX tools and libraries on Windows NT.

Chapter 8

Conclusion

In my opinion this project has gone well. I started out not knowing even whether the language and system features I wanted were possible to implement in the time available — I think I have proved that they were. Whilst the language and runtime system are by no means perfect they have proved themselves to be useful and relatively easy to understand. Of course the language syntax and semantics would need to be defined much more rigidly if it were to be released, and the runtime system would need a large amount of tidying up.

The evaluation given in Section 7 indicates that there is scope for many improvements, both to the language and to its implementation, so the project has not reached a 'dead end'. In particular I feel that the keyword changes in Figure 7.2 would be beneficial. Also, porting to other platforms is important.

8.1 Acknowledgements

A few people other than myself have had an impact on my project.

Jeff Lee wrote the *Lex* and *Yacc* specifications that I used as a basis for the *Distributed C* compiler.

Andrew Main seems to be an expert on the C language and parsing it. Talking to him helped convince me that parsing types fully was not as difficult as it sounded and would greatly simplify other code. I talked about many of my proposed language designs with him.

Roger Packwood was my project supervisor and he acted as both a source of ideas and as someone to discuss the project with. However, I learnt that sometimes it was better to follow my optimism about what was possible rather than his realism, and it seemed to pay off.

Appendix A

Object Descriptions

This section provides full details of the objects used in the compiler. The compiler is described in Section 4.4.

A.1 Token objects

Class: **SourceItem** File: `srcitem.h`

The abstract `SourceItem` class is a direct or indirect base class of every class that can be used in the parse tree or represents an item of source. Its only function declaration (which is a pure virtual function) `GetSourceLine` is used in error reporting to show the line number and original source file for errors and also to maintain the correct line numbers in the output file.

Class: **SourceLine** File: `srcline.h`

The `SourceLine` object is intended to be used as a class member of the other Token objects. It maintains a line number and source filename in an efficient manner. The filename is stored as a pointer into a static set of filenames. The `Compiler` object also uses an instance of this class to maintain the current source filename and line number.

Class: **Token** Base class: **SourceItem** File: `token.h`

The abstract `Token` class is used as a base class for all the token objects. It maintains the token number as used internally by the *Yacc* parser although the current implementation of the compiler makes very little use of this. Its main purpose is to define an interface through pure virtual functions that can be used through a generic `Token *` pointer.

- The `Copy` function is required to make a deep copy of the object without knowing its exact type[1], because of this it must be implemented in each derived class separately. This is useful in the translation of some of the *Distributed C* constructs into C where parts of the code must be duplicated.
- The `GetText` function is used to render the token (recursively if the token is a `ListToken` derivative) into a string ready to be output (or possibly for debugging purposes). It also uses a reference to a `SourceLine` object to insert appropriate line-breaks and `#line` directives in the output.

[1]It would be useful if such a facility were provided by the C++ language.

- If debugging is enabled (the symbol `DEBUG` is defined) then the `DumpTree` function can be used to dump the contents of the token to the debugging device. It is intended to be called cumulatively, so each derived class overrides this function but also calls the base class version.

Class: **TextToken** Base class: **Token** File: `token.h`

The `TextToken` class is used to represent an arbitrary string token in the input. An instance is generally created by the lexer and is passed right through the parser to the output file without modification. `TextToken` objects are used to represent identifiers, keywords, literal strings and multi-character operators. It is possible to retrieve the literal text from a `TextToken` object using the `GetLiteralText` function. The class also provides a static function to turn a `UserType` object into a `TextToken`, this is used when a type name is used as an identifier.

Class: **CharToken** Base class: **Token** File: `token.h`

The `CharToken` class is used in a similar way to `TextToken`, it is really only an optimisation. It is used when the token identifier as maintained by the `Token` class is the ASCII code of the character in the token, this is true for all single character tokens.

Class: **DeclToken** Base class: **Token** File: `token.h`

A `DeclToken` object represents a declaration or a function definition. It is the most complex of the token classes. Each declaration has a number of parts which may or may not be present. Figure A.1 shows a number of declarations with the parts labelled.

1 `StorageClass *pStorage`
 the storage class of the declaration (`extern`, `static`, etc.).
2 `Type *pType`
 the type of the declared object.
3 `MString strIdentifier`
 the name of the declared object.
4 `Token *pInitializer`
 the value to which the object is to be initialised.
5 `Token *pBitfield`
 points to the size in bits when used as part of a structure declaration.
6 `Token *pAttribute`
 the *gcc* `__attribute__` for the declaration. This is require because some of the standard header files include such declarations.[2]
7 `Token *pImplementation`
 points to a statement list for a function definition.

Redundant symbols such as the colon in the bit field or the braces in the implementation aren't actually stored but are recreated when required on output.

The class provides a number of static 'swallow' functions which can be used to create a partially complete `DeclToken` object when it is required in the parser. Many composition functions are also provided to fill in the parts of the declaration and add to them.

The `DeclToken` class also has a number of static functions which are used in the code generation phase to generate the parameter packing/unpacking functions etc.

[2]Various attributes may be added to a declaration to improve the utility of error messages and increase the possibility of optimisation for the compiler. The `__noreturn__` option implies that the function will never return and the `__format__` option allows the arguments to be checked against a `printf` style format string.

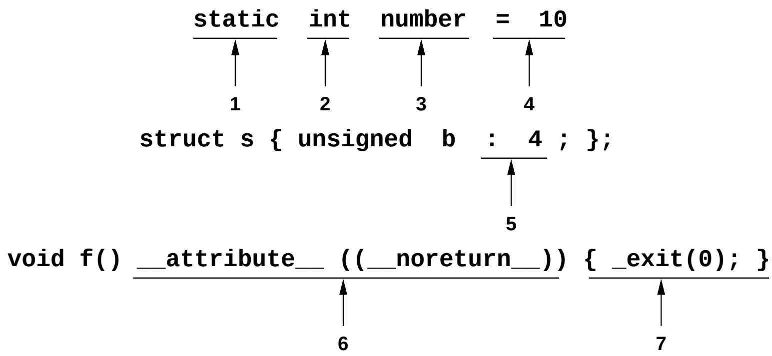


Figure A.1: *Parts of a declaration. 1 – storage class, 2 – type, 3 – identifier, 4 – initialiser, 5 – bit field, 6 – attribute, 7 – implementation.*

Class: **BoundedArrayToken** Base class: **Token** File: `token.h`

A bounded array is an addition to the C language that consists of an array reference and its bound separated by a colon. Neither the array reference nor the bound need to be a single token or constant, they may be function calls. It is used in a `dselect` or `dkill` operation, e.g. `int a[5]; ...; dselect(a:5);`. A `BoundedArrayToken` contains two pointers to `Token` objects (or their derivatives), one for the array expression and one for the bound expression. If the bound expression is *NULL* then the object represents a single variable being used in a context where a bounded array is allowed, this is supported.

Class: **RemoteStartToken** Base class: **Token** File: `token.h`

A `RemoteStartToken` object represents a remote function invocation for the `dfor` and `dblock` constructs. It contains a pointer to a `Token` object which contains the l-value to which the answer is to be assigned, a pointer to a `CompoundToken` object which contains the arguments for the function call, and the name of the function to invoke.

Again the important function is `GetText` which outputs the necessary code to start the remote function.

Class: **DistForToken** Base class: **Token** File: `token.h`

This token contains a complete distributed `for` statement (of the form `dfor(exp1; exp2; exp3) lvalue = rp(args);`) The object contains three pointers to `Token` objects (or their derivatives) for the three expressions and a single pointer to a `RemoteStartToken` for the function to be executed.

The important part of this class is the `GetText` function which returns a string containing proper C code to implement the distributed for statement.

Class: **DistBlockToken** Base class: **Token** File: `token.h`

The `DistBlockToken` class is used to represent a distributed block (of the form `dblock` { `lvalue1 = rp1(args); ...; lvaluen = rpn(args);` }). It contains a single pointer to a `RemoteStartListToken` object for the body.

As with the `DistForToken` class the important function is `GetText` which returns proper C code to implement the distributed block.

Class: **ListToken<T>** Base class: **Token** File: `token.h`

The `ListToken` class is a templated abstract class that handles an arbitrary list of tokens. It uses the C++ standard template library linked list class to do this. Functions are provided to add individual items to the left or right of the list and to concatenate another list to the right.

Class: **CompoundToken** Base class: **ListToken<Token>** File: `token.h`

The `CompoundToken` class is a simple derivation of the `ListToken` class for any type of token (derived from `Token`). It adds a simple `GetText` function which returns the results of calling `GetText` on each token in the list separated by a single space.

`CompoundToken` objects are used throughout the parse tree where further processing of the tokens is not necessary before being output.

Class: **RemoteStartListToken** Base class: **ListToken<RemoteStartToken>** File: `token.h`

The `RemoteStartListToken` class is used to represent a list of `RemoteStartToken` remote function invocation objects for the `DistBlockToken` class. The `GetText` function returns the concatenated results of calling `GetText` on the objects in the list separated by semicolons — this is the actual code that will be compiled by the C compiler.

Class: **DeclListToken** Base class: **ListToken<DeclToken>** File: `token.h`

The `DeclListToken` class maintains a list of declarations. It is used to represent parts of a multiple declaration before the actual type is discovered.

The code `int a, b;` is parsed in two parts. The `int` part is the declaration specifier object, and `a` and `b` are joined into a list of declarators. During the parsing the declarators are formed into a single `DeclListToken` object before the declaration specifier is formed into a `DeclToken` object. Once the complete declaration has been parsed the declaration specifier must be attached to each of the declarators to yield the actual type, this is done by the `DisperseDeclSpecs` static member function. A consequence of this behaviour is that the above declaration will be output as `int a; int b;` in the output file, however the semantics are unchanged.

The object also maintains a separator character and separation method (before, between or after) which are used in the `GetText` function. Because function prototypes are also stored as `DeclListToken` objects an ellipsis can also be added to indicate a variable number of arguments.

Class: **BoundedArrayListToken** Base class: **ListToken<BoundedArrayToken>** File: `token.h`

A `BoundedArrayListToken` object contains a number of bounded array specifications. It is used to represent the arguments for `dselect` and `dkill`.

The `GetText` function, in tandem with `BoundedArrayToken::GetText`, automatically returns the list in the form required for the `_dcx_select` runtime function. For example, `dselect(a:10, b, c:20)` becomes `_dcx_select(3, a, 10, b, 0, c, 20)` with zero indicating that the variable is not an array.

Class: **Type** Base class: **SourceItem** File: `srcitem.h`

The `Type` abstract class provides an interface and some of the functionality required for all types. Since all types have a source line associated with them it contains and manages the `SourceLine` object for all derived classes. Like the `Token` class it provides an interface which must implemented by derived classes:

- The `Copy` function must make a complete deep copy of the object.
- The `Compose` function must compose the type with another. See Section 4.4.5.

- The Render function must correctly add the type to the existing string. This does not necessarily just mean concatenation, for example, a pointer type applied to 'a' would yield '(*a)'. Classes that derive from DerivedType should call its version of Render to ensure that the chain of types is followed.
- If debugging is enabled then the class should define a function called Dump.

Class: **BasicType** Base class: **Type** File: type.h

The BasicType class is used to represent all the C built-in types in signed and unsigned forms. Two enumerated types form constants used to represent the types (void, char, short, int, long, long long, float, double and long double)[3] and the signs (signed, unsigned). Two static arrays contain the text versions of the type names and signs for output. A third static array is used to generate the mangled type names for use in returning a type from a remote function.

Class: **CompoundType** Base class: **Type** File: type.h

A CompoundType object is used to represent a type defined using struct, union or enum. Since such a type may be defined with no name the members strIdentifier and bNamed work together to ensure that this is preserved. The contents of the compound type are stored as a pointer to a DeclListToken object.

Class: **UserType** Base class: **Type** File: type.h

Types defined using typedef are represented by UserType objects. The class is very simple and consists of just the name of the type. The more complicated aspects of user defined types are handled by the lexer and symbol table.

Class: **DerivedType** Base class: **Type** File: type.h

The DerivedType class provides a base class for type modifiers such as pointers and arrays. Its main purpose is to maintain a pointer to the next type or type modifier in the chain and to pass on Render, Mangle and Compose requests.

Class: **PointerType** Base class: **DerivedType** File: type.h

The PointerType class is used to modify the type chain to include a pointer. The class has no significant members since it inherits most of its functionality from DerivedType. A PointerType object is rendered onto an existing type by prepending '(*' and appending ')', the brackets may not always be required but there is no simple way for the object to determine this so they are always included.

Class: **ArrayType** Base class: **DerivedType** File: type.h

An ArrayType object modifies the type chain to include an array of the existing type. Its only significant member is a pointer to a Token object representing the bound expression. An ArrayType is rendered onto an existing type by prepending the qualifiers (if any) and appending the result of calling GetText on the bound expression surrounded by square brackets.

Class: **FunctionType** Base class: **DerivedType** File: type.h

The FunctionType class is used to represent the declarator part of a function prototype (i.e. the argument specification). The parameter list is stored as a pointer to a DeclListToken. The Render

[3] Some of these types are extensions to the ANSI C specification and are included for compatibility with system header files. They will only work if the underlying compiler supports them.

function prepends the qualifiers to the existing type and appends the parameters surrounded by brackets.

Class: **ProcessType** Base class: **DerivedType** File: `type.h`

The `ProcessType` class is used to represent declarations of process type variables (for example, `process<int>`). The important part of a process type is the type on which it is based as specified between the angle brackets, this is the 'inner type' and is stored as a pointer to a generic `Type` object. When a process type is rendered the inner type is mangled and the '`_dcv_`' prefix is added so that the type is actually a pointer to the process variable structure associated with the inner type (.e.g `struct _dcv_i *`).

Class: **PlaceholderType** Base class: **DerivedType** File: `type.h`

A `Placeholder` object is used as a dummy type when parsing an identifier which forms part of a declaration but has unknown type. It is replaced by the real type by the `Type::ReplacePlaceholder` function when the actual type is known.

Class: **Qualifiers** Base class: **SourceItem** File: `srcitem.h`

The `Qualifiers` class is used to represent qualifiers as they are passed from the lexer to the parser and also as part of the `Type` class. An enumerated type provides the constants for each of the qualifiers which are used as bit flags in an integer. In common with tokens the `Qualifiers` class has a `Copy` function and `GetText` function. Any combination of qualifiers applying to the same object are represented by a single instance.

Class: **StorageClass** Base class: **SourceItem** File: `storage.h`

The `StorageClass` class is used to represent combinations of storage class specifiers (`extern`, `static`, etc.). Like the `Qualifiers` class each specifier is represented in an enumeration which is used as a set of bit flags. The class has a `Copy` function and a `Render` function which uses a static array of the text forms of the specifiers. The `ComposeSelf` function allows two `StorageClass` objects to be combined into one.

A.2 Information objects

Class: **FileTable** File: `filetab.h`

A file table is a set of strings that the `SourceLine` class uses to avoid repeating the filename of a token many times. Once a string has been inserted into the set it is not removed until the program terminates, this allows many objects to point at the same string without fear of it being deleted.

The `FileTable` class uses the C++ Standard Template Library `set` class which makes the implementation so trivial it is implemented as inline functions in the header file.

Class: **SymbolKey** File: `syminfo.h`

The `SymbolKey` class is a simple (namespace, symbol) pair which is used as a key to the symbol table. There are six defined namespaces: global, `struct`, `union`, `enum`, generated process variables (`_dcv_` structures) and remote function parameters (`_dcp_` structures). The last two are not really parts of the symbol table but it is used as a convenient means of storing the symbolic information required to avoid repeated declarations.

Class: **SymbolInfo** File: `syminfo.h`

The `SymbolInfo` class is used to hold the actual information that will be held in the symbol table. This consists of the same information stored by a `SymbolKey` object and a numeric scope level which is explained in the description of the `SymbolTable` class. The `HashFunction` function allows a table of `SymbolInfo` objects to be created.

Class: **SymbolTable** File: `syminfo.h`

The `SymbolTable` class implements an important part of the support for the parser. Each time a new type is declared a `SymbolInfo` object is created for it and it is stored in the symbol table.

The `SymbolTable` class is actually just a `typedef` alias for an instance of the `ObjectHash` class specialised to storage `SymbolInfo` objects. The `ObjectHash` class has an extra feature over most hash tables in that it allows multiple entries with the same key. The entries are stored as a linked list forming a stack structure so new entries are *pushed* and *popped*. This feature makes it ideal for storing symbol tables where a type declaration may override one in an outer scope but must disappear at the end of the scope. This procedure is handled by the `Compiler::EnterScope` and `Compiler::LeaveScope` functions.

A.3 Worker objects

Class: **yyFlexLexer** File: `lexer.h`

This class is generated automatically by *Flex*, it encapsulates the tokeniser in a single class. The design of the `FlexLexer` class isn't actually very flexible and *Bison* is incapable of generating a parser class so much of the interface with both is untidy.

Class: **Compiler** File: `compiler.h`

The `Compiler` class is used to control the compilation process. There is only one instance of this object and there may not be more than one due to limitations in the *Bison* generated code. The object holds the symbol table, lexer and I/O streams as well as keeping track of the current input and output line numbers.

Once the command line parameters have been parsed by the `main` function the instance of this class is created and passed the input and output streams. Its first task is to write the preamble file to the output stream (the preamble file contains declarations of the runtime system functions and process structures for the built-in types). I then instructs the parser to parse the input. During the parser's execution many of the functions provided by the `Compiler` object are called, for example the `IsSymbol` and `AddSymbol` are used to access the compiler's symbol table.

A number of code generation functions are declared statically as part of the `Compiler` object. `GenerateRemoteCall`, `GenerateRemoteWait` and `GenerateRemoteSelect` builds the code to implement the `dstart`, `dwait` and `dselect` constructs respectively. The `GenerateDistFor` function performs a similar operation for `dfor`.

When line directives (lines starting with a '`#`' symbol) are read by the lexer they are passed to the `HandleDirective` member function which parses the line number and filename information out of the line and updates the `CurrentLine` member which is an instance of the `SourceLine` class. No standard preprocessor directives should appear in the input file since it has already been passed through the preprocessor.

The `Compiler` class also handles the outputting of error messages and the *Lex/Yacc* function `yyerror` is redefined as a macro to call `Compiler::Error`.

Appendix B

Source Code

The source code for the project is arranged in a number of directories which keep the compiler, runtime system, test programs, etc. separate.

`lib` contains the library source code and header files.

`dc2` contains the compiler source code.

`dcd2` contains the runtime system source code and a simple test program which was used for testing early versions.

`test` contains the test programs for the project.

`tools` contains tools used in the implementation and documentation of the project.

`proto` contains prototype code for investigating many of the techniques used, e.g. shared libraries, threads, etc.

`docs` contains the source for this document and other documentation for the project.

All the source and documentation was held under source control using CVS and the makefiles were generated with the help of GNU Autoconf.

B.1 Mandelbrot Example

B.1.1 test/xmand/xmand.dc

```
#include <stdio.h>
#include <time.h>
#include "xmand.h"

#undef TIMING
#undef CHEAT

double Left = INIT_LEFT;
double Right = INIT_RIGHT;
double Top = INIT_TOP;
double Bottom = INIT_BOTTOM;

#ifdef __dc__
remote
#endif
```

```
chunk mcalc(drect rect, dvect start, int xPos, int yPos)
{
    double dx;
    double dy;
    int x, y;
    chunk c;
    c.xPos = xPos;
    c.yPos = yPos;

    dx = (rect.right - rect.left) / XCHUNK;
    dy = (rect.bottom - rect.top) / YCHUNK;

    for(y = 0; y < YCHUNK; y++)
        for(x = 0; x < XCHUNK; x++)
        {
#ifdef CHEAT
            c.count[y][x] = 1;
#else
            double x0, y0;
            double x1, y1;
            double u, v;
            double r = 0;
            int k = 0;

            x0 = start.x;
            y0 = start.y;

            u = rect.left + x * dx;
            v = rect.top + y * dy;

            while ((r < INFINITY2) && (k < MAXIT))
            {
                x1 = (x0*x0) - (y0*y0) + u;
                y1 = 2.0*x0*y0 + v;
                k++;
                x0 = x1;
                y0 = y1;
                r = (x0*x0) + (y0*y0);
            }
            if (k == MAXIT)
                c.count[y][x] = 0;
            else
                c.count[y][x] = k;
#endif
        }
    return c;
}

#define YBITS (YSIZE/YCHUNK)
#define XBITS (XSIZE/XCHUNK)

#if 0
void map(int image[YSIZE][XSIZE], chunk c)
{
    int x, y;
    for(y = 0; y < YCHUNK; y++)
        for(x = 0; x < XCHUNK; x++)
        {
            image[(c.yPos*YCHUNK) + y][(c.xPos*XCHUNK) + x] = c.count[y][x];
        }
}
#endif

void reposition(int x, int y, int n)
{
    double xorg, yorg;
```

```
    double xsize, ysize;
    xsize = Right - Left;
    ysize = Bottom - Top;

    xorg = (xsize * x) / XSIZE;
    yorg = (ysize * y) / YSIZE;

    switch(n)
    {
    case 1:
        Left += xorg - xsize/4;
        Right += xorg - 3 * xsize/4;
        Top += yorg - ysize/4;
        Bottom += yorg - 3 * ysize/4;
        break;
    case 2:
        Left += xorg - xsize/2;
        Right += xorg - xsize/2;
        Top += yorg - ysize/2;
        Bottom += yorg - ysize/2;
        break;
    case 3:
        Left += xorg - xsize;
        Right += xorg;
        Top += yorg - ysize;
        Bottom += yorg;
        break;
    }
}

void manddraw()
{
#ifdef __dc__
    process<chunk> results[YBITS * XBITS];
    int j;
#else
    chunk results[YBITS * XBITS];
#endif
#ifdef TIMING
    time_t timeStart, timeStop;
#endif
    int x, y;
    double dx, dy;

    dvect start = { XSTART, YSTART };

    dx = (Right - Left) / XBITS;
    dy = (Bottom - Top) / YBITS;

#ifdef TIMING
    time(&timeStart);
#endif

    for(y = 0; y < YBITS; y++)
        for(x = 0; x < XBITS; x++)
        {
            drect r;
            r.left = Left + x * dx;
            r.right = Left + (x+1) * dx;
            r.top = Top + y * dy;
            r.bottom = Top + (y+1) * dy;
#ifndef TIMING
            markpending(x, y);
#endif
#ifdef __dc__
            results[y * XBITS + x] = dstart mcalc(r, start, x, y);
```

```
#else
            results[y * XBITS + x] = mcalc(r, start, x, y);
#ifndef TIMING
            drawchunk(&results[y * XBITS + x]);
#endif
#endif
        }

#ifdef __dc__
    for(j = 0; j < YBITS * XBITS; j++)
    {
        int i;
        chunk c;
        i = dselect(results:(XBITS*YBITS));
        c = dwait(results[i]);
#ifndef TIMING
        drawchunk(&c);
#endif
    }
#endif

#ifdef TIMING
    time(&timeStop);
    printf("Time taken: %ld seconds.\n", timeStop - timeStart);
#endif

#ifdef OUTPUT
    /* Output */
    printf("P3\n%d %d %d\n", XSIZE, YSIZE, 255);
    for(y = 0; y < YSIZE; y++)
        for(x = 0; x < XSIZE; x++)
        {
            colour c = colours[image[y][x] % COLOURS];
            printf("%d %d %d\n", RED(c), GREEN(c), BLUE(c));
        }
#endif
}
```

B.1.2 test/xmand/xwin.c

```
/* X Window Stuff */

#include <stdio.h>
#include <X11/X.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/cursorfont.h>
#include "xmand.h"

static Atom wmDeleteWindow;

colour colours[] =
{
    RGB(0, 0, 0),
    RGB(127, 0, 0),
    RGB(255, 0, 0),
    RGB(255, 127, 0),
    RGB(127, 128, 0),
    RGB(127, 255, 0),
    RGB(0, 255, 0),
    RGB(0, 255, 127),
    RGB(0, 255, 255),
    RGB(0, 127, 255),
    RGB(0, 0, 255),
    RGB(127, 0, 255),
```

```
    RGB(255, 0, 255),
    RGB(255, 0, 127),
    RGB(0, 127, 0),
    RGB(0, 0, 127)
};

#define COLOURS (sizeof(colours)/sizeof(colour))

#define BORDER_WIDTH 4

long cindex[COLOURS];
Colormap colourmap;

Display *display;
Window window;
GC gc;

int screen;

int main(int argc, char **argv)
{
    const char *szName = "Distributed C Mandelbrot Viewer";
    XSizeHints hints;
    Cursor crosscursor;
    Cursor waitcursor;
    int done = 0;
    int drawing = 1;
    XEvent event;
    XSetWindowAttributes attr;

    if (!(display = XOpenDisplay("")))
    {
        fprintf(stderr, "Couldn't open display, giving up\n");
        return 1;
    }
    screen = DefaultScreen(display);

    printf("Opened display\n");
    fflush(stdout);

    if (!alloccolours())
    {
        fprintf(stderr, "Couldn't allocate colours\n");
        return 2;
    }
    hints.flags = PSize | PMinSize | PMaxSize;
    hints.height = YSIZE;
    hints.width = XSIZE;
    hints.x = 0;
    hints.y = 0;
    hints.min_height = YSIZE;
    hints.min_width = XSIZE;
    hints.max_height = YSIZE;
    hints.max_width = XSIZE;

    window = XCreateSimpleWindow(display, DefaultRootWindow(display),
                                 hints.x, hints.y, hints.width,
                                 hints.height, BORDER_WIDTH,
                                 WhitePixel(display, screen),
                                 BlackPixel(display, screen));

    crosscursor = XCreateFontCursor (display, XC_crosshair);
    waitcursor = XCreateFontCursor (display, XC_watch);

    attr.backing_store = Always;
    attr.cursor = waitcursor;
```

```
    XChangeWindowAttributes (display, window, CWBackingStore|CWCursor, &attr);

    gc = XCreateGC(display, window, 0, 0);

    XSetStandardProperties(display, window, szName, szName, None, argv,
                         argc, &hints);

    wmDeleteWindow = XInternAtom (display, "WM_DELETE_WINDOW", False);
    XSetWMProtocols (display, window, &wmDeleteWindow, 1);

    XSelectInput(display, window,
                 ExposureMask|ButtonPressMask|VisibilityChangeMask);

    XMapWindow(display, window);

    XSelectInput(display, window, ButtonPressMask);

    XSetForeground(display, gc, BlackPixel(display, screen));
    XFillRectangle(display, window, gc, 0, 0, XSIZE, YSIZE);

    printf("Accepting events...\n");
    fflush(stdout);
    while (!done)
    {
        while(XPending(display))
        {
            XNextEvent(display, &event);
            switch (event.type)
            {
            case Expose:
                break;
            case ButtonPress:
                /*printf("Button %d pressed at %d, %d\n",
                       event.xbutton.button,
                       event.xbutton.x,
                       event.xbutton.y);*/
                reposition(event.xbutton.x, event.xbutton.y,
                           event.xbutton.button);
                drawing = 1;
                break;
            case ClientMessage:
                if (event.xclient.format == 32
                    && event.xclient.data.l[0] == wmDeleteWindow)
                    done = 1;
                break;

            }
        }

        if (drawing)
        {
            attr.cursor = waitcursor;
            XChangeWindowAttributes (display, window, CWCursor, &attr);
            manddraw();
            XBell(display, 50);
            drawing = 0;
            attr.cursor = crosscursor;
            XChangeWindowAttributes (display, window, CWCursor, &attr);
        }
    }

    XFreeCursor(display, crosscursor);
    XFreeCursor(display, waitcursor);

    return 0;
}
```

```
void drawpixel(int x, int y, long l)
{
    XSetForeground(display, gc,
                   cindex[l % COLOURS]);
    XDrawPoint(display, window, gc, x, y);
}

void drawchunk(chunk *pc)
{
    int x, y;
    int xorg, yorg;
    xorg = pc->xPos * XCHUNK;
    yorg = pc->yPos * YCHUNK;

    for(y = 0; y < YCHUNK; y++)
        for(x = 0; x < XCHUNK; x++)
        {
            XSetForeground(display, gc, cindex[pc->count[y][x] % COLOURS]);
            XDrawPoint(display, window, gc, x + xorg, y + yorg);
        }
    flushwindow();
}

void flushwindow()
{
/*  XCopyArea(display, window, window, gc, 0, 0, XSIZE, YSIZE, 0, 0);*/
    XFlush(display);
}

int alloccolours()
{
    int i;
    XColor cell;
    colourmap = XDefaultColormap(display, screen);

    for(i = 0; i < COLOURS; i++)
    {
        char szBuffer[10];
        sprintf(szBuffer, "#%02xd%02xd%02xd",
                RED(colours[i]),
                GREEN(colours[i]),
                BLUE(colours[i]));

        if (!XParseColor(display,colourmap,szBuffer,&cell))
            return 0;
        else
            if (!XAllocColor(display, colourmap, &cell))
                return 0;
        cindex[i] = cell.pixel;
    }
    return 1;
}

void markpending(int x, int y)
{
    XSetForeground(display, gc, WhitePixel(display, screen));
    XDrawRectangle(display, window, gc, x * XCHUNK, y * YCHUNK,
                   XCHUNK-1, YCHUNK-1);
    XFlush(display);
}
```

B.1.3 test/xmand/xmand.h

```
#ifndef XMAND_H
#define XMAND_H 1

#if 1
#define XSIZE 800
#define YSIZE 800

#define XCHUNK 50
#define YCHUNK 50
#else
#define XSIZE 256
#define YSIZE 256

#define XCHUNK 128
#define YCHUNK 128
#endif

#define XSTART 0.0
#define YSTART 0.0

#if 0
#define INIT_LEFT   (-0.875)
#define INIT_RIGHT  (-0.8625)
#define INIT_TOP    (0.2375)
#define INIT_BOTTOM (0.25)
#define INFINITY2 (10.1)
#else
#define INIT_LEFT   (-1.8)
#define INIT_RIGHT  (0.6)
#define INIT_TOP    (-1.2)
#define INIT_BOTTOM (1.2)
#define INFINITY2 (10.1)
#endif

#define MAXIT 50

typedef struct
{
    double left;
    double top;
    double right;
    double bottom;
} drect;

typedef struct
{
    double x;
    double y;
} dvect;

typedef unsigned long colour;

#define RGB(R, G, B) ((R) + ((G) << 8) + ((B) << 16))

#define RED(C) ((unsigned char)((C) & 255))
#define GREEN(C) ((unsigned char)(((C) >> 8) & 255))
#define BLUE(C) ((unsigned char)(((C) >> 16) & 255))

typedef struct
{
    int count[YCHUNK][XCHUNK];
    int xPos;
    int yPos;
} chunk;
```

```
void drawchunk(chunk *);
void drawpixel(int, int, long);
void flushwindow();
int alloccolours();
void manddraw();
void markpending(int, int);
void reposition(int, int, int);

extern colour colours[];

extern double Left;
extern double Right;
extern double Top;
extern double Bottom;

#endif
```

B.2 Addition Example

B.2.1 test/add/add.dc

```
#include <stdio.h>

remote double add(double, double);

int main()
{
    process<double> pa,pb;
    double ra, rb;
    int i, j;

    printf("Starting processes\n");

    pa = dstart add(3.0, 7.2);
    pb = dstart add(4.7, 12.23);

    i = 2;
    while (i)
    {
        j = dselect(pa, pb);
        printf("Return from dselect = %d\n", j);
        switch (j)
        {
        case 0:
            ra = dwait(pa);
            printf("Result from A = %f\n", ra);
            --i;
            break;
        case 1:
            rb = dwait(pb);
            printf("Result from B = %f\n", rb);
            --i;
            break;
        default:
            printf("Something odd happened, dselect returned %d\n", j);
        }
    }
    printf("Everything is finished\n");
    return 0;
}

remote double add(double a, double b)
{
```

```
    return (a + b);
}
```

B.3 Factorial Example

B.3.1 test/rfac/rfac.dc

```
#include <stdio.h>
#include <stdlib.h>

#ifdef __dc__
#define RFAC(A) dwait(dstart rfac(A))
#else
#define remote
#define RFAC(A) rfac(A)
#endif

remote double rfac(double n);

int main(int ac, char *av[])
{
    long n;
    if (ac != 2)
    {
        fprintf(stderr, "Must supply argument\n");
        return 1;
    }
    n = atol(av[1]);
    if (n < 1)
    {
        fprintf(stderr, "Argument must be strictly positive\n");
        return 2;
    }

    printf("%ld! = %-40.0f\n", n, RFAC(n));

    return 0;
}

remote double rfac(double n)
{
    return (n == 1) ? 1 : n * RFAC(n-1);
}
```

B.4 Overhead Example

B.4.1 test/overhead/oh.dc

```
/* Overhead Example */
#include <time.h>
#include <stdio.h>

#define COUNT 200
#define DELAY 50000

#ifndef __dc__
#define remote
#endif
```

```
/* In microseconds */
void mysleep(long l)
{
    struct timespec t;
    t.tv_sec = l / 1000000;
    t.tv_nsec = (l % 1000000) * 1000;

    /*printf("Sleeping for %ld/%ld\n", (long)t.tv_sec, (long)t.tv_nsec);*/
    nanosleep(&t, NULL);
}

remote int dolittle(long a)
{
    mysleep(a);
    return 0;
}

int main()
{
#ifdef __dc__
    process<int> p;
#endif
    int a, i;
    time_t timeStart, timeStop;

    time(&timeStart);
    for(i = 0; i < COUNT; i++)
    {
#ifdef __dc__
        p = dstart dolittle(DELAY);
        a = dwait(p);
#else
        a = dolittle(DELAY);
#endif
    }
    time(&timeStop);

    printf("Time: %ld seconds\n", timeStop - timeStart);
    return 0;
}
```

B.5 Sample Test Programs

B.5.1 dc2/s1.dc

```
/* Sample One
 * ==========
 * Simple sample program which calls the remote function 'bing' to render
 * the value and twice the value of a double variable into a string which
 * is then printed.
 */

#include <stdio.h>

/* Can't handle strings longer than 255 characters. */
struct lstring
{
    char sz[256];
};

/* Declare the function as callable remotely. */
remote struct lstring bing(double d);
```

```
int main()
{
    process<struct lstring> a;
    struct lstring i;
    a = dstart bing(3.141);
    i = dwait(a);
    printf("%s\n", i.sz);

    return 0;
}

remote struct lstring bing(double d)
{
    struct lstring a;
    sprintf(a.sz, "%f, %f\n", d, 2.0*d);
    return a;
}
```

B.5.2 dc2/s2.dc

```
/* Sample Two
 * ==========
 * This program starts NUM threads which perform a calculation and then
 * sleep for a random amount of time. The thread results are output in the
 * order that they finish.
 */

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <float.h>

#define NUM 24

remote int bing(double d);

int main()
{
    process<int> a[NUM];
    int i;
    int n;
    int j;

    for(i = 0; i < NUM; i++)
        a[i] = dstart bing(i + 69.0);

    printf("All threads started\n");
    for(i = 0; i < NUM; i++)
    {
        n = dselect(a:NUM);
        printf("Got a finished thread, index: %d\n", n);
        j = dwait(a[n]);
        printf("%d: %d: %d\n", i, n, j);
    }
    return 0;
}

void mysleep(long l)
{
    struct timespec t;
    t.tv_sec = l / 1000000;
    t.tv_nsec = (l % 1000000) * 1000;
```

```
    printf("Sleeping for %ld/%ld\n", (long)t.tv_sec, (long)t.tv_nsec);
    nanosleep(&t, NULL);
}

remote int bing(double d)
{
    int i = 0;
    while (d > (1 + DBL_EPSILON))
        i++, d = sqrt(d);
    mysleep(rand() % 1000000);
    return i;
}
```

B.5.3 dc2/s3.dc

```
/* Sample Three
 * ============
 * A simple example of the use of the distributed for statement.
 */

#include <stdio.h>

#define COUNT 25

remote int bing(int i);

int main()
{
    int i;
    int a[COUNT];
    dfor(i = 0; i < COUNT; i++)
        a[i] = bing(i);

    for(i = 0; i < COUNT; i++)
        printf("%d\n", a[i]);
    return 0;
}

remote int bing(int i)
{
    return i * i * i;
}
```

B.5.4 dc2/s4.dc

```
/* Sample Four
 * ===========
 * Example of the distributed block structure.
 */

#include <stdio.h>

remote int bing(int i);

int main()
{
    int i;
    int a, b, c;
    dblock {
        a = bing(1);
        b = bing(2);
```

```
        c = bing(3);
    }
    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("c = %d\n", c);
    return 0;
}

remote int bing(int i)
{
    return i * i * i;
}
```

B.6 Root Directory

B.6.1 Makefile.in

```
# Makefile.in for project/

srcdir = @srcdir@
VPATH = @srcdir@
TOP = .
THIS= .

SUBDIRS=lib proto dc2 dcd2 test docs
SUBDIRSR=docs test dcd2 dc2 proto lib

include $(TOP)/Make.rules

all : dep bin doc COL

bin :
	for i in $(SUBDIRS); do $(MAKE) -C $$i bin; done

dep :
	for i in $(SUBDIRS); do $(MAKE) -C $$i dep; done

install :
	for i in $(SUBDIRS); do $(MAKE) -C $$i install; done

doc : doc_me
	for i in $(SUBDIRS); do $(MAKE) -C $$i doc; done

clean : clean_me
	for i in $(SUBDIRSR); do $(MAKE) -C $$i clean; done

distclean : clean_me
	for i in $(SUBDIRSR); do $(MAKE) -C $$i distclean; done
	$(RM) Makefile Make.rules config.cache config.log config.status

spotless : distclean
	$(RM) configure


clean_me :
	$(RM) *

doc_me : pretty.tex

pretty.tex : Makefile.in Make.rules.in configure.in
	$(TEXIFY) $^ > pretty.tex

COL : Makefile.in configure.in acconfig.h Make.rules.in
```

```
	$(80COL) $^

Makefile : Makefile.in $(TOP)/config.status
	cd $(TOP) && CONFIG_FILES=$(THIS)/Makefile CONFIG_HEADERS= ./config.status
```

B.6.2 Make.rules.in

```
.SUFFIXES:

LEX=flex
YACC=bison
CC = gcc
CXX = g++
RM = rm -f
LDXX = g++
LD = gcc
DEBUG = @DEBUG@
INSTALL = @INSTALL@
80COL = $(TOP)/tools/80col
TEXIFY = $(TOP)/tools/texify
ETAGS = etags

INCLUDE=$(TOP)/lib/include
LIBDIR=$(TOP)/lib

prefix=@prefix@
exec_prefix=@exec_prefix@
bindir=@bindir@
datadir=@datadir@

#LEXDEBUG=-d
#YACCDEBUG=-v

LD_DYNAMIC = @LD_DYNAMIC@
LIBPROJECT= -lproject
LIBEFENCE= @LIBEFENCE@
LIBPOSIX4= @LIBPOSIX4@
LIBDL = @LIBDL@
LOADLIBES= -lg++ -lproject
#DEP=$(CC) -M -I$(INCLUDE)
DEP=@$(TOP)/tools/mkdep -I$(INCLUDE)
LEXFLAGS = $(LEXDEBUG) -+
YACCFLAGS = $(YACCDEBUG) --defines -v

PROJLIB=$(LIBDIR)/libproject.a

LIBPTHREAD = @LIBPTHREAD@
LIBPOSIX4 = @LIBPOSIX4@
LIBSOCKET = @LIBSOCKET@
LIBX11    = @LIBX11@

CFLAGS_THREAD = -D_REENTRANT
CXXFLAGS_THREAD = $(CFLAGS_THREAD)
LDFLAGS_THREAD =

CFLAGS_SHARED = -fpic
CXXFLAGS_SHARED = $(CFLAGS_SHARED)
LDFLAGS_SHARED = -shared

LIBS=-lproject
LIBDEPS = $(PROJLIB)

CFLAGS = -c -Wall -I$(INCLUDE)
CXXFLAGS = -c -Wall -I$(INCLUDE)
```

```
LDFLAGS = -L$(LIBDIR)

ifneq ($(DEBUG), 0)
CFLAGS += -g -DDEBUG
CXXFLAGS += -ggdb -DDEBUG
LDFLAGS += -g
LIBS += $(LIBEFENCE)
else
CFLAGS += -O2
CXXFLAGS += -O2
LDFLAGS += -s
endif

%.o : %.C ; $(CXX) $(CXXFLAGS) -c -o $@ $<
%.s : %.C ; $(CXX) $(CXXFLAGS) -S -o $@ $<
%.o : %.c ; $(CC) $(CFLAGS) -c -o $@ $<
%.s : %.c ; $(CC) $(CFLAGS) -c -o $@ $<
%.lex.C : %.l   ; $(LEX) $(LEXFLAGS) -o$*.lex.C $<
%.tab.C : %.y   ; $(YACC) $(YACCFLAGS) -o $*.tab.c -b $*.tab $<; \
          mv $*.tab.c $*.tab.C

# Cancel some built in rules
%.o : %.l
%.c : %.l
%.C : %.l
%.o : %.y
%.c : %.y
%.C : %.y
```

B.6.3 configure.in

```
dnl Process this file with autoconf to produce a configure script.
AC_INIT(dcd/main.c)
AC_CONFIG_HEADER(lib/include/config.h)
AC_CANONICAL_HOST

dnl Handle debugging
echo $ac_n "checking if we are debugging... $ac_c" 1>&6
AC_ARG_ENABLE(debug, [Enable debugging support], DEBUG=1, DEBUG=0)
export DEBUG
AC_SUBST(DEBUG)
if [[ "$DEBUG" = "0" ]]; then
 echo "no" 1>&6
else
 echo "yes" 1>&6
fi

dnl Checks for programs.
AC_PROG_CC
AC_PROG_LEX
AC_PROG_YACC
AC_PROG_MAKE_SET
AC_PROG_INSTALL
AC_PROG_CPP

dnl Checks for libraries.
AC_CHECK_LIB(dl, dlopen, LIBDL="-ldl")
AC_SUBST(LIBDL)
AC_CHECK_LIB(posix4, nanosleep, LIBPOSIX4="-lposix4")
AC_SUBST(LIBPOSIX4)
AC_CHECK_LIB(socket, socket, LIBSOCKET="$LIBSOCKET -lsocket")
AC_CHECK_LIB(nsl, gethostbyname, LIBSOCKET="$LIBSOCKET -lnsl")
AC_CHECK_LIB(resolv, herror, LIBSOCKET="$LIBSOCKET -lresolv")
AC_SUBST(LIBSOCKET)
```

```
AC_CHECK_LIB(pthread, pthread_create, LIBPTHREAD="-lpthread")
AC_SUBST(LIBPTHREAD)
LIBS="$LIBS $LIBPOSIX4 $LIBSOCKET"

dnl LIBTHREAD=
dnl AC_CHECK_FUNCS(pthread_create)
dnl if eval "test \"$ac_cv_func_pthread_create\" = no"; then
dnl  AC_CHECK_LIB(pthread, pthread_create, LIBTHREAD=-lpthread)
dnl  if eval "test \"$ac_cv_lib_pthread_pthread_create\" = no"; then
dnl   AC_CHECK_LIB(pthreads, pthread_create, LIBTHREAD=-lpthreads)
dnl   if eval "test \"$ac_cv_lib_pthreads_pthread_create\" = no"; then
dnl    echo "*" 1>&6
dnl    echo "* Can't cope without POSIX threads, sorry" 1>&6
dnl    echo "*" 1>&6
dnl   fi
dnl  fi
dnl fi
dnl AC_SUBST(LIBTHREAD)

AC_CHECK_LIB(efence, EF_ALIGNMENT, LIBEFENCE="-lefence")
AC_SUBST(LIBEFENCE)

savedLDFLAGS=$LDFLAGS
LDFLAGS="$LDFLAGS -L/usr/openwin/lib"
AC_CHECK_LIB(X11, XOpenDisplay, LIBX11="-L/usr/openwin/lib -lX11")
AC_SUBST(LIBX11)

dnl Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS(fcntl.h unistd.h pthread.h stdlib.h limits.h dlfcn.h string.h)
AC_CHECK_HEADERS(sys/param.h sys/ioctl.h sys/time.h sys/types.h sys/socket.h)
AC_CHECK_HEADERS(sys/stat.h sys/utsname.h sys/resource.h)
AC_CHECK_HEADERS(netinet/in.h netdb.h)

dnl Checks for typedefs, structures, and compiler characteristics.
AC_C_CONST
AC_TYPE_PID_T
AC_HEADER_TIME
AC_TYPE_SIZE_T

dnl Checks for library functions.
AC_FUNC_ALLOCA
AC_FUNC_MEMCMP
AC_FUNC_VPRINTF
AC_CHECK_FUNCS(getcwd gettimeofday socket strdup strstr nanosleep)
AC_CHECK_FUNCS(uname setsid fork dup dup2 gethostbyname setrlimit)

dnl Do my own custom check for -rdynamic
echo $ac_n "checking if we need -rdynamic""... $ac_c" 1>&6
case "$host_os" in
linux*)     LD_DYNAMIC=-rdynamic ; echo "yes" 1>&6;;
*)  echo "no" 1>&6;;
esac
AC_SUBST(LD_DYNAMIC)dnl

dnl Do my own custom check for CONCURRENT
echo $ac_n "checking if we can only do concurrent""... $ac_c" 1>&6
case "$host_os" in
linux*)     CONCURRENT=1 ; echo "yes" 1>&6;;
*)  CONCURRENT=0 ; echo "no" 1>&6;;
esac
AC_DEFINE_UNQUOTED(CONCURRENT, $CONCURRENT)dnl

eval preamble_path=$datadir/dc2.preamble
AC_DEFINE_UNQUOTED(PREAMBLEFILE, "$preamble_path")
```

```
AC_OUTPUT(Make.rules Makefile \
      lib/Makefile lib/src/Makefile lib/test/Makefile lib/include/Makefile\
      docs/Makefile docs/oral/Makefile docs/demo/Makefile dc2/Makefile \
      dcd2/Makefile dcd2/test/Makefile \
      proto/Makefile proto/expr/Makefile proto/ansic/Makefile \
      proto/dyn/Makefile proto/threads/Makefile \
      test/Makefile test/overhead/Makefile \
      test/xmand/Makefile test/rfac/Makefile test/add/Makefile )
```

B.7 Compiler

B.7.1 dc2/Makefile.in

```
# Makefile.in for project/dc2/dc2

srcdir = @srcdir@
VPATH = @srcdir@
TOP = ..
THIS=dc2

include $(TOP)/Make.rules

SRC = dc2.C compiler.C token.C type.C storage.C compose.C codegen.C qual.C \
      srcline.C
OBJ = dc.tab.o dc.lex.o $(SRC:.C=.o)
TARGET = dc2

HEADERS = dc2.h compiler.h lexer.h syminfo.h token.h type.h storage.h \
      qual.h

PREAMBLEPATH = $(datadir)/dc2.preamble
SAMPLEPATH = $(datadir)/samples/

TESTS = test[0123456789].dc s[0123456789].dc

all : bin

bin : $(TARGET) pseudo

clean : clean.tests
    $(RM) $(TARGET) $(OBJ) *  dc.tab.h dc.lex.C dc.tab.C dc.output .depend
    $(RM) pretty.tex

clean.tests :
    $(RM) test[123456789].di test[123456789].gen.c test[123456789].so
    $(RM) s[123456789].di s[123456789].gen.c s[123456789].so s[123456789].o

distclean : clean
    $(RM) Makefile

doc : pretty samples

install : install.dirs install.bin install.data

install.dirs :
    $(INSTALL) -d $(bindir)
    $(INSTALL) -d $(datadir)
    $(INSTALL) -d $(SAMPLEPATH)

install.bin : $(TARGET)
    $(INSTALL) $(TARGET) $(bindir)
    $(INSTALL) dcc $(bindir)
```

```
install.data :
	$(INSTALL) dc2.preamble $(PREAMBLEPATH)
	$(INSTALL) s1.dc $(SAMPLEPATH)
	$(INSTALL) s2.dc $(SAMPLEPATH)
	$(INSTALL) s3.dc $(SAMPLEPATH)

pseudo : TAGS COL

pretty : Makefile.in dc2.preamble $(SRC) $(HEADERS) dc.y dc.l dcc dc2seqc
	$(TEXIFY) $^ > pretty.tex

samples : s1.dc s2.dc s3.dc s4.dc
	$(TEXIFY) $^ > samples.tex

TAGS : $(SRC) dc.y dc.l
	$(ETAGS) $^

COL : $(SRC) dc.y dc.l $(TESTS) Makefile.in dc2.preamble
	$(80COL) $^

%.gen.c : %.dc
	dcc -d -Wall -g $< $(LIBPOSIX4)

%.so : %.gen.c
	dcc -o$*.so -Wall -g $< $(LIBPOSIX4)

$(TARGET) : $(OBJ) $(LIBDEPS)
	$(LDXX) $(LDFLAGS) -o $@ $(OBJ) $(LIBS)

dc.lex.o : *.h dc.tab.C dc.lex.C $(INCLUDE)/*.h *.h
dc.tab.h : dc.tab.C
dc.tab.o : *.h $(INCLUDE)/*.h

.depend : $(SRC) $(HEADERS) $(INCLUDE)/*.h
	$(DEP) $(SRC) > .depend

Makefile : Makefile.in $(TOP)/config.status
	cd $(TOP) && CONFIG_FILES=$(THIS)/Makefile CONFIG_HEADERS= ./config.status

-include .depend
```

B.7.2 dc2/dc2.preamble

```
struct _dch_invoke
{
    unsigned long cbPayload;
    char szName[64];
};

struct _dch_var
{
    unsigned long cbPayload;
    void *tid;
    int status;
    unsigned int qpos;
};

typedef void *_dct_bucket;

extern int _dcx_exec(const char *, struct _dch_var *, struct _dch_invoke *);
extern int _dcx_wait(struct _dch_var *);
extern int _dcx_setfn(struct _dch_invoke *, const char *);
extern int _dcx_select(int, ...);

extern void *_dcx_allocvar(unsigned int);
```

```
extern void *_dcx_allocargs(unsigned int);
extern void *_dcx_alloc(unsigned int);
extern void _dcx_free(void *);

extern _dct_bucket _dcx_bucket_create();
extern void _dcx_bucket_add(_dct_bucket, void *, struct _dch_var *);
extern void _dcx_bucket_wait(_dct_bucket);

extern char _dc_ObjectName[];

struct _dcv_c  { struct _dch_var hdr; char val; };
struct _dcv_Sc { struct _dch_var hdr; signed char val; };
struct _dcv_Uc { struct _dch_var hdr; unsigned char val; };
struct _dcv_s  { struct _dch_var hdr; short val; };
struct _dcv_Ss { struct _dch_var hdr; signed short val; };
struct _dcv_Us { struct _dch_var hdr; unsigned short val; };
struct _dcv_i  { struct _dch_var hdr; int val; };
struct _dcv_Si { struct _dch_var hdr; signed int val; };
struct _dcv_Ui { struct _dch_var hdr; unsigned int val; };
struct _dcv_l  { struct _dch_var hdr; long val; };
struct _dcv_Sl { struct _dch_var hdr; signed long val; };
struct _dcv_Ul { struct _dch_var hdr; unsigned long val; };
struct _dcv_L  { struct _dch_var hdr; long long val; };
struct _dcv_SL { struct _dch_var hdr; signed long long val; };
struct _dcv_UL { struct _dch_var hdr; unsigned long long val; };
struct _dcv_f  { struct _dch_var hdr; float val; };
struct _dcv_d  { struct _dch_var hdr; double val; };
struct _dcv_D  { struct _dch_var hdr; long double val; };
```

B.7.3 dc2/dc2.C

```
#include "project.h"
#include "dc2.h"
#include "lexer.h"

#define new DEBUG_NEW

int main(int ac, char *av[])
{
    if (ac != 2)
    {
        fprintf(stderr, "Things are not hoopy\n");
        exit(1);
    }
    MString strInputFile = av[1];
//  MString strOutputFile = strInputFile;
//  if (strInputFile.Right(3) == ".dc")
//      strOutputFile.DropRight(3);
//  strOutputFile += ".c";

//  printf("Input file = %s, Output file = %s\n", (const char *)strInputFile,
//         (const char *)strOutputFile);

    if (strInputFile == "-")
    {
        Compiler c(cin, cout, strInputFile);
        return c.GetError();
    }
    else
    {
        ifstream fileInput;

        fileInput.open(strInputFile, ios::in);
        if (!fileInput)
        {
```

```
            cerr << "Failed to open input file: \'"
                 << strInputFile << "\'" << endl;
            return 1;
        }
        Compiler c(fileInput, cout, strInputFile);
        return c.GetError();
    }
}
```

B.7.4 dc2/compiler.C

```
#include "project.h"
#include <sys/stat.h>
#include "dc2.h"
#include "compiler.h"
#include "dc.tab.h"

#define new DEBUG_NEW

Compiler *Compiler::pInstance = NULL;

Compiler::Compiler(istream &i, ostream &o, const char *s)
    : symtab(91), fileInput(i), fileOutput(o), strInputFile(s), nError(0),
      CurrentLine(1, s), OutputLine(1, s)
{
    ASSERT(pInstance == NULL);
    pInstance = this;
    nError = 0;
    nCurrentScope = 0;

    if (FindPreambleFile())
    {
        lexer.switch_streams(&fileInput);

        OutputPreamble();
        yyparse();
    }
    else
    {
        Error("Unable to find preamble file\n");
        nError = 2;
    }

//  CompleteOutput();
}

Compiler:: Compiler()
{
}

Compiler *Compiler::Instance()
{
    return pInstance;
}

void Compiler::Error(const char *s)
{
#if 1
    cerr << CurrentLine.GetFile() << ":" << CurrentLine.GetLine() << ":";
    if (lexer.YYText())
        cerr << s << " near \'" << lexer.YYText() << "\'" << endl;
    else
        cerr << s << " near null symbol." << endl;
#else
    cerr << s << endl;
```

```
#endif
}

bool Compiler::IsSymbol(Namespace ns, const char *s)
{
    return (symtab.IsMember(SymbolKey(ns, s)));
}

void Compiler::AddSymbol(Namespace ns, const char *s)
{
    SymbolInfo i(ns, s, nCurrentScope);
    symtab.Add(i);
}

void Compiler::EnterScope()
{
    nCurrentScope++;
}

void Compiler::LeaveScope()
{
    // Remove all the stuff from the typetable that was local to the current
    // scope

    for(SymbolTableIterator i = symtab; i; i++)
    {
        if (symtab[i].GetScopeLevel() == nCurrentScope)
        {
            symtab.Remove(i);
        }
    }

    nCurrentScope--;
}

void Compiler::Count()
{
    int i;
    const char *yytext = lexer.YYText();

    for (i = 0 ; yytext[i] != '\0'; i++)
        if (yytext[i] == '\n')
            CurrentLine.Advance();
}

void Compiler::OutputPreamble()
{
#ifdef PREAMBLE
    DeclListToken *p = DeclToken::GeneratePreambleDecls();
    Output(p->GetText());
    delete p;
#else
//  FindPthreadFile();

//  IncludeFile(strPthreadFile);
    IncludeFile(strPreambleFile);
#endif
}

inline bool FileExists(const char *s)
{
    struct stat i;
    return (stat(s, &i) == 0);
}

bool Compiler::FindPreambleFile()
```

```
{
    if (getenv("DC_PREAMBLE"))
    {
        strPreambleFile = getenv("DC_PREAMBLE");
        if (FileExists(strPreambleFile))
            return true;
        cerr << "DC_PREAMBLE environment variable set incorrectly" << endl;
    }

    if (getenv("HOME"))
    {
        strPreambleFile = getenv("HOME");
        strPreambleFile += "/share/dc2.preamble";
        if (FileExists(strPreambleFile))
            return true;
    }
    else
        cerr << "You are homeless" << endl;

    strPreambleFile = PREAMBLEFILE;
    return (FileExists(strPreambleFile));
}

bool Compiler::FindPthreadFile()
{
    strPthreadFile = "/usr/local/include/pthread.h";
    if (FileExists(strPthreadFile))
        return true;

    strPthreadFile = "/usr/include/pthread.h";
    if (FileExists(strPthreadFile))
        return true;

    return false;
}


void Compiler::IncludeFile(const char *s)
{
    ifstream filePreamble;
    filePreamble.open(s, ios::in);
    if (!filePreamble)
    {
        cerr << "Failed to open file: \'"
             << s << "\'" << endl;
    }
    else
    {
        fileOutput << "# 1 \"" << strPreambleFile << "\"" << endl;
        char ch;
        while((ch = filePreamble.get()) != EOF)
        {
            fileOutput << ch;
        }
//      fileOutput << "# 1 \"" << strInputFile << "\"" << endl;
    }
}

void Compiler::HandleDirective(const char *s)
{
    MString str = s;
    int nLine;
    MString strFile;

    // Strip off whitespace
    str.LTrim();
```

```
    str.RTrim();
    ASSERT(str[0] == '#');
    str.DropLeft();
    str.LTrim();

    if (str.Left(4) == "line")
    {
        str.DropLeft(4);
        str.LTrim();
    }
    sscanf(str, "%d %s", &nLine, strFile.Grab(str.GetLength()));
    strFile.Release();
    if (strFile[0] == '\"')
    {
        ASSERT(strFile.Right(1) == "\"");
        strFile.DropLeft();
        strFile.DropRight();
    }
//  TRACE1("Line: %s\n", (const char *)str);
//  TRACE2("File: %s, line: %d\n", (const char *)strFile, nLine);
    Compiler::Instance()->CurrentLine.Set(nLine, strFile);
}

void Compiler::Output(const Token *p)
{
//  TRACE2("Doing output on outputline %s:%d\n", OutputLine.GetFile(),
//         OutputLine.GetLine());
    fileOutput << p->GetText(OutputLine);
}
```

B.7.5 dc2/token.C

```
#include "project.h"
#include "compiler.h"
#include "token.h"
#include "type.h"
#include "tokid.h"

#define new DEBUG_NEW

//
// class Token
//

Token::Token(int n)
    : nTokenId(n)
{
//  TRACE1("Token @%p\n", this);
}

Token::Token(const Token &t)
{
    nTokenId = t.nTokenId;
}

Token:: Token()
{
}

int Token::GetTokenId() const
{
    return nTokenId;
}

#ifdef DEBUG
```

```
void Token::DumpTree(int nDepth) const
{
    ASSERT(this != NULL);
    if (nDepth == 0)
        TRACE0("### Dumping tree\n");
    SourceLine sl = SourceLine::Null();
    TRACE4("DUMP: [%p] %*s %s\n", this, nDepth, "",
           (const char *)GetText(sl));
}
#endif

DeclToken *Token::GetDeclToken()
{
    return NULL;
}

//
// class ListToken<T> : Token
//

template<class T>
ListToken<T>::ListToken(int n)
    : Token(n)
{
}

template<class T>
ListToken<T>::ListToken(const ListToken &t)
    : Token(t)
{
    for(TokenList::const_iterator i = t.listChildren.begin();
        i != t.listChildren.end(); ++i)
    {
        AdoptRight((*i)->Copy());
    }
}

template<class T>
ListToken<T>:: ListToken()
{
    for(TokenList::iterator i = listChildren.begin();
        i != listChildren.end(); ++i)
    {
//      listChildren.erase(i);
        delete *i;
    }
}

template<class T>
void ListToken<T>::AdoptLeft(T *p)
{
//  TRACE2("Adopting node on left %p: %d\n", p, p->GetTokenId());
    if (p)
        listChildren.push_front(p);
}

template<class T>
void ListToken<T>::AdoptRight(T *p)
{
//  TRACE2("Adopting node on right %p: %d\n", p, p->GetTokenId());
    if (p)
        listChildren.push_back(p);
}

#ifdef DEBUG
template<class T>
```

```
void ListToken<T>::DumpTree(int nDepth) const
{
    Token::DumpTree(nDepth);

    // Now do the children
    ++nDepth;

    for(TokenList::const_iterator i = listChildren.begin();
        i != listChildren.end(); ++i)
    {
        (*i)->DumpTree(nDepth);
    }
}
#endif

template<class T>
void ListToken<T>::ConcatRight(ListToken<T> *p)
{
    if (p)
    {
//      TRACE1("Concatting: %s\n", (const char *)GetText());
//      TRACE1("With      : %s\n", (const char *)p->GetText());

#ifdef DEBUG
        TokenList::size_type size = listChildren.size()
                                    + p->listChildren.size();
#endif
        listChildren.splice(listChildren.end(), p->listChildren);

        ASSERT(p->listChildren.size() == 0);
        ASSERT(size == listChildren.size() + p->listChildren.size());

        delete p;
    }
}

template<class T>
unsigned int ListToken<T>::Count() const
{
    return listChildren.size();
}

template<class T>
const SourceLine &ListToken<T>::GetSourceLine() const
{
    return (*(listChildren.begin()))->GetSourceLine();
}

//
// class CompoundToken : Token
//

CompoundToken::CompoundToken(int n)
    : ListToken<Token>(n)
{
//  TRACE1("Created compound token: %s\n", (const char *)GetTokenName(this));
}

CompoundToken::CompoundToken(const CompoundToken &t)
    : ListToken<Token>(t)
{
}

CompoundToken *CompoundToken::Copy() const
{
    return new CompoundToken(*this);
```

```
}

MString CompoundToken::GetText(SourceLine &sl) const
{
    MString str;
//  TRACE1("CompoundToken::GetText Iterating through compound at %p\n", this);
    for(TokenList::const_iterator i = listChildren.begin();
        i != listChildren.end(); ++i)
    {
        str += (*i)->GetText(sl);
        str += ' ';
    }
//  TRACE1("Compounded text = %s\n", (const char *)str);
    return str;
}

#ifdef DEBUG
void CompoundToken::DumpTree(int nDepth) const
{
    TRACE0("A CompoundToken:\n");
    ListToken<Token>::DumpTree(nDepth);
}
#endif

CompoundToken:: CompoundToken()
{
}


//
// class TextToken : public Token
//

TextToken::TextToken(const SourceLine &sl, int n, const char *s)
    : Token(n), line(sl), strText(s)
{
//  TRACE2("Created TextToken(%s, %s)\n", (const char *)GetTokenName(this), s);
}

TextToken::TextToken(const TextToken &t)
    : Token(t), line(t.line), strText(t.strText)
{
}

TextToken *TextToken::Copy() const
{
    return new TextToken(*this);
}

MString TextToken::GetText(SourceLine &sl) const
{
//  TRACE1("TextToken::GetText() returning %s\n", (const char *)strText);
    MString str = line.LineCorrect(sl);
    str += strText;
    sl.Set(line);
    return str;
}

const char *TextToken::GetLiteralText() const
{
    return strText;
}

TextToken *TextToken::SwallowType(UserType *pType)
{
    TextToken *p;
```

```
    p = new TextToken(pType->GetSourceLine(), TYPE_NAME, pType->GetName());
    delete pType;
    return p;
}

const SourceLine &TextToken::GetSourceLine() const
{
    return line;
}

//
// class CharToken : public Token
//

CharToken::CharToken(const SourceLine &sl, char ch)
    : Token(ch), line(sl)
{
}

CharToken::CharToken(const CharToken &t)
    : Token(t), line(t.line)
{
}

CharToken *CharToken::Copy() const
{
    return new CharToken(*this);
}

MString CharToken::GetText(SourceLine &sl) const
{
    MString str = line.LineCorrect(sl);
    str += (char)GetTokenId();
    sl.Set(line);
    return str;
}

const SourceLine &CharToken::GetSourceLine() const
{
    return line;
}

//
// class DeclToken : public Token
//

DeclToken::DeclToken(const SourceLine &sl, int n, StorageClass *psc, Type *pt,
                     const char *s)
    : Token(n), line(sl), pStorage(psc), pType(pt), strIdentifier(s),
      pInitializer(NULL), pBitfield(NULL), pImplementation(NULL),
      pAttribute(NULL)
{
}

DeclToken::DeclToken(const DeclToken &t)
    : Token(t), line(t.line)
{
//  TRACE0("Copying DeclToken\n");
    pStorage = t.pStorage->Copy();

    if (t.pType)
        pType = t.pType->Copy();
    else
        pType = NULL;

    if (t.pBitfield)
```

```
        pBitfield = t.pBitfield->Copy();
    else
        pBitfield = NULL;

    if (t.pInitializer)
        pInitializer = t.pInitializer->Copy();
    else
        pInitializer = NULL;

    if (t.pImplementation)
        pImplementation = t.pImplementation->Copy();
    else
        pImplementation = NULL;

    if (t.pAttribute)
        pAttribute = t.pAttribute->Copy();
    else
        pAttribute = NULL;

    strIdentifier = t.strIdentifier;
}

DeclToken *DeclToken::Copy() const
{
    return new DeclToken(*this);
}

DeclToken:: DeclToken()
{
//  TRACE2("DeclToken going away (%p) %s\n", this, (const char *)GetText());
    ASSERT(pStorage);
    delete pStorage;
    delete pType;
    delete pInitializer;
    delete pBitfield;
    delete pImplementation;
    delete pAttribute;
}

DeclToken *DeclToken::GetDeclToken()
{
    return this;
}

MString DeclToken::GetText(SourceLine &sl) const
{
    MString str;

    pStorage->Render(sl, str);

    str += line.LineCorrect(sl);
    sl.Set(line);

    MString str2 = strIdentifier;

    if (pType)
        pType->Render(sl, str2);

    str += str2;

    if (pInitializer)
    {
        str += '=';
        str += pInitializer->GetText(sl);
    }
```

```
    if (pBitfield)
    {
        str += ':';
        str += pBitfield->GetText(sl);
    }

    if (pImplementation)
    {
        str += pImplementation->GetText(sl);
    }

    if (pAttribute)
    {
        str += ' ';
        str += pAttribute->GetText(sl);
    }

    return str;
}

void DeclToken::ComposeStorageClass(StorageClass *psc)
{
    pStorage->ComposeSelf(psc);
    delete psc;
}

void DeclToken::ComposeType(Type *p)
{
//  TRACE2("Composing types: %p, %p\n", pType, p);
    if (pType)
    {
//      TRACE0("Composing inside DeclToken\n");
        pType = p->Compose(pType);
    }
    else
        pType = p;
}

void DeclToken::ComposeQualifiers(Qualifiers *pq)
{
    pType->ComposeQualifiers(pq);
}

void DeclToken::ComposeSelf(DeclToken *p)
{
    if (p->pType)
    {
        ComposeType(p->pType);
        p->pType = NULL;
    }
    pStorage->ComposeSelf(p->pStorage);
    if (strIdentifier.GetLength())
    {
        if (p->strIdentifier.GetLength())
            Error("Duplicate identifiers in declaration.");
    }
    else
        strIdentifier = p->strIdentifier;

    delete p;
}

DeclToken *DeclToken::SwallowType(Type *p, const char *s)
{
    const SourceLine &sl = p->GetSourceLine();
    return new DeclToken(sl, DECLSPEC, new StorageClass(sl), p, s);
```

```
}

DeclToken *DeclToken::SwallowStorageClass(StorageClass *psc)
{
//  TRACE0("DeclToken swallowing storage class\n");
    return new DeclToken(psc->GetSourceLine(), DECLSPEC, psc, NULL, "");
}

DeclToken *DeclToken::SwallowQualifiers(Qualifiers *pq)
{
    const SourceLine &sl = pq->GetSourceLine();
    Type *pType = new BasicType(sl, pq, BasicType::T_NONE,
                                BasicType::T_NOSIGN);

    return new DeclToken(sl, DECLSPEC, new StorageClass(sl),
                         pType, "");
}

DeclToken *DeclToken::SwallowIdentifier(const SourceLine &sl, const char *s)
{
    return new DeclToken(sl, IDENTIFIER, new StorageClass(sl), NULL, s);
}

DeclToken *DeclToken::SwallowBitfield(Token *t)
{
    const SourceLine &sl = t->GetSourceLine();
    DeclToken *p = new DeclToken(sl, DECLSPEC, new StorageClass(sl), NULL, "");
    p->SetBitfield(t);
    return p;
}

DeclToken *DeclToken::CreatePlaceholder(const SourceLine &sl, const char *s)
{
    Type *pType = new PlaceholderType(sl);
    return new DeclToken(sl, DECLSPEC, new StorageClass(sl), pType, s);
}

#ifdef DEBUG
void DeclToken::DumpTree(int nDepth) const
{
    TRACE0("##Dumping DeclToken START\n");
    Token::DumpTree(nDepth);

    if (pType)
        pType->Dump();
    else
        TRACE0("No type\n");
    TRACE0("##Dumping DeclToken END\n");
}
#endif

CompoundToken *DeclToken::FindParmList()
{
//  return pType->FindParmList();
    return NULL;
}

void DeclToken::AdjoinBasicType(DeclToken *t)
{
    ASSERT(t->strIdentifier.GetLength() == 0);

    pStorage->ComposeSelf(t->pStorage);

//  BasicType *pb = t->pType->GetBasicType();
//  ASSERT(pb);
```

```
    if (pType)
        pType = pType->ReplacePlaceholder(t->pType);
    else
        pType = t->pType;

    t->pType = NULL;

//  TRACE1("Removing token at %p after adjoin\n", t);
    delete t;
}

void DeclToken::SetInitializer(Token *t)
{
    ASSERT(pInitializer == NULL);
    pInitializer = t;
}

void DeclToken::SetImplementation(Token *t)
{
    ASSERT(pImplementation == NULL);
    pImplementation = t;
}

void DeclToken::SetBitfield(Token *t)
{
    ASSERT(pBitfield == NULL);
    pBitfield = t;
}

void DeclToken::SetAttribute(Token *t)
{
    ASSERT(pAttribute == NULL);
    pAttribute = t;
}

void DeclToken::ChangeIdentifier(const char *s)
{
    strIdentifier = s;
}

const char *DeclToken::GetIdentifier() const
{
    return strIdentifier;
}

Type *DeclToken::ExtractType()
{
    Type *p = pType;
    pType = NULL;
    return p;
}

// If asked, we don't want separators on functions that have
// implementations.

bool DeclToken::IsSeparatorNeeded()
{
    return (pImplementation == NULL);
}

const SourceLine &DeclToken::GetSourceLine() const
{
    return line;
}

//
```

```
// class DeclListToken : ListToken<DeclToken>
//

DeclListToken::DeclListToken(int n, char c, int f)
    : ListToken<DeclToken>(n), cSeparator(c), nSeparatorPos(f),
      bEllipsis(false)
{
}

DeclListToken::DeclListToken(const DeclListToken &t)
    : ListToken<DeclToken>(t), cSeparator(t.cSeparator),
      nSeparatorPos(t.nSeparatorPos), bEllipsis(t.bEllipsis)
{
}

DeclListToken:: DeclListToken()
{
}

DeclListToken *DeclListToken::Copy() const
{
    return new DeclListToken(*this);
}

DeclListToken *DeclListToken::DisperseDeclSpecs(DeclToken *d, DeclListToken *p,
                                                char cSep, int n)
{
    DeclListToken *pOut = new DeclListToken(STMTLIST, cSep, n);

    TokenList::iterator pCurrent, pNext;

//  TRACE1("Disperse [%p]\n", d);

#ifdef DEBUG
    TokenList::size_type nSize = p->listChildren.size();
    ASSERT(pOut->listChildren.size() == 0);
#endif
    pNext = p->listChildren.begin();
    while (pNext != p->listChildren.end())
    {
        pCurrent = pNext;
        ++pNext;
        DeclToken *pNewDecl = d->Copy();
        DeclToken *pCurrentDecl = *pCurrent;
//      TRACE1("New decl is [%p]\n", pNewDecl);
//      TRACE1("Current decl is [%p]\n", pCurrentDecl);

        if (pCurrentDecl)
        {
            pCurrentDecl->AdjoinBasicType(pNewDecl);
            pOut->AdoptRight(pCurrentDecl);
        }
        else
        {
            TRACE0("No DeclToken, must be a separator\n");
            ASSERT(false);
        }
        p->listChildren.erase(pCurrent);
    }

//  TRACE2("Original: %d, New: %d\n", nSize, pOut->listChildren.size());

    ASSERT(nSize == pOut->listChildren.size());
    delete p;
    delete d;
```

```
//  TRACE1("Dispersed: %s\n", (const char *)pOut->GetText());
    return pOut;
}


void DeclListToken::AddEllipsis()
{
    ASSERT(bEllipsis == false);
    bEllipsis = true;
}

void DeclListToken::SetSeparator(char c, int n)
{
    cSeparator = c;
    nSeparatorPos = n;
}

MString DeclListToken::GetText(SourceLine &sl) const
{
    bool bBetween = (nSeparatorPos == S_POST) || (nSeparatorPos == S_BETWEEN);
    MString str;
    if (nSeparatorPos == S_PRE)
        str += cSeparator;

    for(TokenList::const_iterator i = listChildren.begin();
        i != listChildren.end(); ++i)
    {
        str += (*i)->GetText(sl);
        if (bBetween || ((nSeparatorPos == S_ASK) &&
                         ((*i)->IsSeparatorNeeded())))
            str += cSeparator;
    }
    if (bEllipsis)
    {
        str += "...";
        if (nSeparatorPos == S_POST)
            str += cSeparator;
    }
    else
    {
        if (nSeparatorPos == S_BETWEEN)
            str.DropRight();
    }

//  TRACE2("DeclList GetText(%d) = '%s'\n", nSeparatorPos, (const char *)str);
    return str;
}

#ifdef DEBUG
void DeclListToken::DumpTree(int nDepth) const
{
    ListToken<DeclToken>::DumpTree(nDepth);
}
#endif

DeclListToken *DeclListToken::SwallowDeclToken(DeclToken *p, char c, int s)
{
    DeclListToken *pNew = new DeclListToken(DECLSPEC, c, s);
    pNew->AdoptRight(p);
    return pNew;
}

void DeclListToken::ConcatRight(DeclListToken *p)
{
    if (p)
    {
```

```
        ASSERT(nSeparatorPos == p->nSeparatorPos);
        ASSERT(cSeparator == p->cSeparator);
        ASSERT(!(bEllipsis && p->bEllipsis));
    }
    ListToken<DeclToken>::ConcatRight(p);
}

//
// class BoundedArrayToken : Token
//

BoundedArrayToken::BoundedArrayToken(const SourceLine &sl, Token *p, Token *q)
    : Token(GENTOK), line(sl), pExpression (p), pBound(q)
{
}

BoundedArrayToken::BoundedArrayToken(const BoundedArrayToken &t)
    : Token(t), line(t.line), pExpression(t.pExpression), pBound(t.pBound)
{
}

BoundedArrayToken:: BoundedArrayToken()
{
    delete pExpression;
    delete pBound;
}

BoundedArrayToken *BoundedArrayToken::Copy() const
{
    return new BoundedArrayToken(*this);
}

MString BoundedArrayToken::GetText(SourceLine &sl) const
{
    MString s = line.LineCorrect(sl);
    sl.Set(line);
    if (pBound)
        s += pBound->GetText(sl);
    else
        s += "0";

    s += ',';

    if (pExpression)
        s += pExpression->GetText(sl);
    else
        s += "(0)";

    return s;
}

const SourceLine &BoundedArrayToken::GetSourceLine() const
{
    return line;
}

//
// class BoundedArrayListToken : ListToken<BoundedArrayToken>
//

BoundedArrayListToken::BoundedArrayListToken(int n)
    : ListToken<BoundedArrayToken>(n)
{
}

BoundedArrayListToken::BoundedArrayListToken(const BoundedArrayListToken &t)
```

```
    : ListToken<BoundedArrayToken>(t)
{
}

BoundedArrayListToken:: BoundedArrayListToken()
{
}

BoundedArrayListToken *BoundedArrayListToken::Copy() const
{
    return new BoundedArrayListToken(*this);
}

MString BoundedArrayListToken::GetText(SourceLine &sl) const
{
    MString s;

    sprintf(s.Grab(256), "%d,", Count());
    s.Release();

    for(TokenList::const_iterator i = listChildren.begin();
        i != listChildren.end(); ++i)
    {
        s += (*i)->GetText(sl);
        s += ',';
    }

    s.DropRight();

    //TRACE1("<<%s>>\n", (const char *)s);
    return s;
}

BoundedArrayListToken *
BoundedArrayListToken::SwallowBoundedArray(BoundedArrayToken *p)
{
    BoundedArrayListToken *pNew = new BoundedArrayListToken(GENTOK);
    pNew->AdoptRight(p);
    return pNew;
}

#ifdef DEBUG
void BoundedArrayListToken::DumpTree(int nDepth) const
{
    TRACE0("A BoundedArrayListToken\n");
    ListToken<BoundedArrayToken>::DumpTree(nDepth);
}
#endif

//
// class RemoteStartToken : Token
//

RemoteStartToken::RemoteStartToken(const SourceLine &sl, const char *s,
                                   CompoundToken *pp, Token *pa)
    : Token(GENTOK), line(sl), strIdentifier(s), pArgs(pp), pAssign(pa)
{
}

RemoteStartToken::RemoteStartToken(const RemoteStartToken &t)
    : Token(t), line(t.line), strIdentifier(t.strIdentifier),
      pArgs(NULL), pAssign(NULL)
{
    if (t.pArgs)
        pArgs = t.pArgs->Copy();
    if (t.pAssign)
```

```
        pAssign = t.pAssign->Copy();
}

RemoteStartToken:: RemoteStartToken()
{
    delete pArgs;
    delete pAssign;
}

RemoteStartToken *RemoteStartToken::Copy() const
{
    return new RemoteStartToken(*this);
}

MString RemoteStartToken::GetText(SourceLine &sl) const
{
    MString s = line.LineCorrect(sl);
    sl.Set(line);

    if (pAssign)
    {
        s += "_dcx_bucket_add(_dct_b, &(";
        s += pAssign->GetText(sl);
        s += "), (struct _dch_var *)";
    }
    s += "_dcf_";
    s += strIdentifier;
    s += '(';
    s += pArgs->GetText(sl);
    s += ')';
    if (pAssign)
        s += ");";
    return s;
}

const SourceLine &RemoteStartToken::GetSourceLine() const
{
    return line;
}

RemoteStartToken *RemoteStartToken::Create(TextToken *pi,
                                           CompoundToken *pp, Token *pa)
{
    RemoteStartToken *p;
    p = new RemoteStartToken(pi->GetSourceLine(), pi->GetLiteralText(),
                             pp, pa);
    delete pi;
    return p;
}

const char *RemoteStartToken::GetIdentifier()
{
    return strIdentifier;
}

const CompoundToken *RemoteStartToken::GetArgs()
{
    return pArgs;
}

const Token *RemoteStartToken::GetAssigned()
{
    return pAssign;
}

//
```

```
// class RemoteStartListToken : ListToken<RemoteStartToken>
//

RemoteStartListToken::RemoteStartListToken(int n)
    : ListToken<RemoteStartToken>(n)
{
}

RemoteStartListToken::RemoteStartListToken(const RemoteStartListToken &t)
    : ListToken<RemoteStartToken>(t)
{
}

RemoteStartListToken *RemoteStartListToken::Copy() const
{
    return new RemoteStartListToken(*this);
}

MString RemoteStartListToken::GetText(SourceLine &sl) const
{
    MString s;

    for(TokenList::const_iterator i = listChildren.begin();
        i != listChildren.end(); ++i)
    {
        s += (*i)->GetText(sl);
        s += ';';
    }

    s.DropRight();

    return s;
}

RemoteStartListToken *RemoteStartListToken::Swallow(RemoteStartToken *pt)
{
    RemoteStartListToken *p = new RemoteStartListToken(GENTOK);
    p->AdoptRight(pt);
    return p;
}

//
// class DistForToken : public Token
//

DistForToken::DistForToken(const SourceLine &sl,
                           Token *pe1, Token *pe2, Token *pe3,
                           RemoteStartToken *pb)
    : Token(DFOR), line(sl), pExp1(pe1), pExp2(pe2), pExp3(pe3), pBody(pb)
{
}

DistForToken::DistForToken(const DistForToken &t)
    : Token(t), line(t.line),
      pExp1(NULL), pExp2(NULL), pExp3(NULL), pBody(NULL)
{
    if (t.pExp1)
        pExp1 = t.pExp1->Copy();
    if (t.pExp2)
        pExp2 = t.pExp2->Copy();
    if (t.pExp3)
        pExp3 = t.pExp3->Copy();
    if (t.pBody)
        pBody = t.pBody->Copy();
}
```

```
DistForToken:: DistForToken()
{
    delete pExp1;
    delete pExp2;
    delete pExp3;
    delete pBody;
}

DistForToken *DistForToken::Copy() const
{
    return new DistForToken(*this);
}

MString DistForToken::GetText(SourceLine &sl) const
{
    ASSERT(pExp1);
    ASSERT(pExp2);
    ASSERT(pExp3);
    ASSERT(pBody);
    ASSERT(pBody->GetAssigned);

    MString s = line.LineCorrect(sl);
    sl.Set(line);

    s += '{';
    s += "_dct_bucket _dct_b = _dcx_bucket_create();";
    s += "for(";
    if (pExp1)
        s += pExp1->GetText(sl);
    if (pExp2)
        s += pExp2->GetText(sl);
    if (pExp3)
        s += pExp3->GetText(sl);
    s += ')';
    s += pBody->GetText(sl);
    s += "_dcx_bucket_wait(_dct_b);}";

    return s;
}

const SourceLine &DistForToken::GetSourceLine() const
{
    return line;
}

//
// class DistBlockToken : public Token
//

DistBlockToken::DistBlockToken(const SourceLine &sl,
                           RemoteStartListToken *pb)
    : Token(DFOR), line(sl), pBody(pb)
{
}

DistBlockToken::DistBlockToken(const DistBlockToken &t)
    : Token(t), line(t.line),
      pBody(NULL)
{
    if (t.pBody)
        pBody = t.pBody->Copy();
}

DistBlockToken:: DistBlockToken()
{
    delete pBody;
```

```
}

DistBlockToken *DistBlockToken::Copy() const
{
    return new DistBlockToken(*this);
}

MString DistBlockToken::GetText(SourceLine &sl) const
{
    MString s = line.LineCorrect(sl);
    sl.Set(line);

    if (pBody)
    {
        s += '{';
        s += "_dct_bucket _dct_b = _dcx_bucket_create();";
        s += pBody->GetText(sl);
        s += "_dcx_bucket_wait(_dct_b);";
        s += '}';
    }
    return s;
}

const SourceLine &DistBlockToken::GetSourceLine() const
{
    return line;
}

template class list<Token *>;
template class list<DeclToken *>;
template class ListToken<Token>;
template class ListToken<DeclToken>;
```

B.7.6 dc2/type.C

```
#include "project.h"
#include "token.h"
#include "type.h"

#define new DEBUG_NEW

//
// class Type
//

Type::Type(const SourceLine &sl)
    : line(sl)
{
    pQual = new Qualifiers(sl);
//  TRACE1("Created quals at %p\n", pQual);
}

Type::Type(const SourceLine &sl, Qualifiers *q)
    : line(sl), pQual(q)
{
//  TRACE1("Inherited quals at %p\n", pQual);
}

Type:: Type()
{
//  TRACE1("Waving goodbye to quals at %p\n", pQual);
    ASSERT(pQual);
    delete pQual;
//  TRACE1("Type going away: %p\n", this);
}
```

```
Type::Type(const Type &t)
    : line(t.line)
{
    ASSERT(t.pQual);
    pQual = t.pQual->Copy();
//  TRACE2("Copied quals at %p to %p\n", t.pQual, pQual);
}

MString Type::GetQualifiers(SourceLine &sl) const
{
    ASSERT(pQual);
    return pQual->GetText(sl);
}

void Type::ComposeQualifiers(Qualifiers *pq)
{
    pQual->Merge(pq);
//  TRACE1("Merged quals at %p, now deleting\n", pq);
    delete pq;
}

void Type::ComposeQualifiers(Type *p)
{
    pQual->Merge(p->pQual);
}

BasicType *Type::GetBasicType()
{
    return NULL;
}

DerivedType *Type::GetDerivedType()
{
    return NULL;
}

FunctionType *Type::GetFunctionType()
{
    return NULL;
}

Type *Type::ReplacePlaceholder(Type *)
{
    ASSERT(false);
}

Type *Type::GetDefinitionType()
{
    return NULL;
}

MString Type::MangleIdentifier(const char *i)
{
    char szBuffer[(sizeof(size_t) * CHAR_BIT)/3 + 2];
    sprintf(szBuffer, "%d", strlen(i));
    MString s = szBuffer;
    s += i;
    return s;
}

void Type::Mangle(MString &str) const
{
    ASSERT(pQual);
    pQual->Mangle(str);
}
```

```
const SourceLine &Type::GetSourceLine() const
{
    return line;
}

MString Type::LineCorrect(SourceLine &sl) const
{
    MString str = line.LineCorrect(sl);
    sl.Set(line);
    return str;
}

//
// class BasicType : Type
//

const char *BasicType::szBasicMangles[] =
{
    "", "v", "c", "s", "i", "l", "L", "f", "d", "D"
};

const char *BasicType::szSignMangles[] =
{
    "", "S", "U"
};

const char *BasicType::szBasicTypes[] =
{
    "/*none*/",
    "void ",
    "char ",
    "short ",
    "int ",
    "long ",
    "long long ",
    "float ", "double ", "long double "
};

const char *BasicType::szBasicSigns[] =
{
    "",
    "signed ",
    "unsigned "
};

BasicType::BasicType(const SourceLine &sl, Qualifiers *pq, int n, int s)
    : Type(sl, pq), nSimpleType(n), nSign(s)
{
    ASSERT(n >= 0);
    ASSERT(n < NUMBER_OF_TYPES);
    ASSERT(s >= 0);
    ASSERT(s < NUMBER_OF_SIGNS);
//  TRACE3("Created BasicType(%d, %s, %d)\n", q, szBasicTypes[n], s);
}

BasicType::BasicType(const SourceLine &sl, int n, int s)
    : Type(sl), nSimpleType(n), nSign(s)
{
    ASSERT(n >= 0);
    ASSERT(n < NUMBER_OF_TYPES);
    ASSERT(s >= 0);
    ASSERT(s < NUMBER_OF_SIGNS);
//  TRACE3("Created BasicType(%d, %s, %d)\n", q, szBasicTypes[n], s);
}
```

```
BasicType::BasicType(const BasicType &t)
    : Type(t), nSimpleType(t.nSimpleType), nSign(t.nSign)
{
}

BasicType *BasicType::Copy() const
{
    return new BasicType(*this);
}

void BasicType::Render(SourceLine &sl, MString &str) const
{
//  TRACE0("Rendering basic type\n");
    MString str2;
    str2 += GetQualifiers(sl);
    str2 += LineCorrect(sl);
    str2 += szBasicSigns[nSign];
    str2 += szBasicTypes[nSimpleType];
    str = str2 + str;
}

void BasicType::Mangle(MString &str) const
{
    Type::Mangle(str);
    str += szSignMangles[nSign];
    str += szBasicMangles[nSimpleType];
}

BasicType *BasicType::GetBasicType()
{
    return this;
}

#ifdef DEBUG
void BasicType::Dump() const
{
    TRACE2("DUMP: [%p] Basic Type: %s\n", this, szBasicTypes[nSimpleType]);
}
#endif

//
// class CompoundType : Type
//

CompoundType::CompoundType(const SourceLine &sl, Qualifiers *pq, int n,
                           const char *s, DeclListToken *p, bool b)
    : Type(sl, pq), nClass(n), strName(s), pContents(p), bNamed(b)
{
//  TRACE3("Created CompoundType(%d, %d, %s,", q, n, s);
//  TRACE2("%p, %d)\n", p, b);
}

CompoundType::CompoundType(const SourceLine &sl, int n, const char *s,
                           DeclListToken *p, bool b)
    : Type(sl), nClass(n), strName(s), pContents(p), bNamed(b)
{
//  TRACE3("Created CompoundType(%d, %d, %s,", q, n, s);
//  TRACE2("%p, %d)\n", p, b);
}

// Don't generally want to copy the contents of the type, jut its name
CompoundType::CompoundType(const CompoundType &t)
    : Type(t), nClass(t.nClass), strName(t.strName), pContents(NULL),
      bNamed(t.bNamed)
{
    if (t.pContents)
```

```
        pContents = t.pContents->Copy();
    else
        pContents = NULL;
}

CompoundType:: CompoundType()
{
    delete pContents;
}

CompoundType *CompoundType::Copy() const
{
    return new CompoundType(*this);
}

void CompoundType::Render(SourceLine &sl, MString &str) const
{
//  TRACE1("Rendering compound type onto '%s'\n", (const char *)str);

    MString pre = LineCorrect(sl);

    pre += GetQualifiers(sl);

    switch (nClass)
    {
    case C_STRUCT:
        pre += "struct ";
        break;
    case C_UNION:
        pre += "union ";
        break;
    case C_ENUM:
        pre += "enum ";
        break;
    default:
        ASSERT(false);
        break;
    }

    if (bNamed)
    {
        pre += strName;
        pre += ' ';
    }

    if (pContents)
    {
        pre += "{";
        pre += pContents->GetText(sl);
        pre += "}";
    }

    str = pre + str;
}

void CompoundType::Mangle(MString &str) const
{
    Type::Mangle(str);
    switch(nClass)
    {
    case C_STRUCT:
    case C_UNION:
        str += 'G';
    }
    str += Type::MangleIdentifier(strName);
}
```

```
void CompoundType::SetName(const char *s)
{
    ASSERT(strName.GetLength() == 0);
    strName = s;
    bNamed = true;
}

void CompoundType::SetNameless()
{
    ASSERT(strName.GetLength() == 0);
    strName = "__unnamed__";
    bNamed = false;
}

void CompoundType::SetContents(DeclListToken *p)
{
    ASSERT(pContents == NULL);
    pContents = p;
}

#ifdef DEBUG
void CompoundType::Dump() const
{
    const char *pre;
    switch (nClass)
    {
    case C_STRUCT:
        pre = "struct ";
        break;
    case C_UNION:
        pre = "union ";
        break;
    case C_ENUM:
        pre = "enum ";
        break;
    default:
        ASSERT(false);
        break;
    }

    SourceLine sl = SourceLine::Null();
    MString s;
    if (pContents)
        s = pContents->GetText(sl);
}
#endif

Type *CompoundType::GetDefinitionType()
{
    if (bNamed && pContents)
        return new CompoundType(GetSourceLine(), nClass, strName, NULL, true);
    else
        return NULL;
}

//
// class UserType : Type
//

UserType::UserType(const SourceLine &sl, Qualifiers *pq, const char *s)
    : Type(sl, pq), strName(s)
{
//  TRACE2("Created UserType(%d, %s)\n", q, s);
}
```

```
UserType::UserType(const SourceLine &sl, const char *s)
    : Type(sl), strName(s)
{
//  TRACE2("Created UserType(%d, %s)\n", q, s);
}

UserType::UserType(const UserType &t)
    : Type(t), strName(t.strName)
{
}

UserType *UserType::Copy() const
{
    return new UserType(*this);
}

void UserType::Render(SourceLine &sl, MString &str) const
{
//  TRACE0("Rendering user type\n");
    MString str2 = LineCorrect(sl);

    str2 += GetQualifiers(sl);
    str2 += strName;
    str2 += ' ';
    str = str2 + str;
}

void UserType::Mangle(MString &str) const
{
    str += 'T';
    str += Type::MangleIdentifier(strName);
}

const char *UserType::GetName() const
{
    return strName;
}

#ifdef DEBUG
void UserType::Dump() const
{
    TRACE2("DUMP: [%p] User Type: %s\n", this, (const char *)strName);
}
#endif

//
// class DerivedType : Type
//

DerivedType::DerivedType(const SourceLine &sl, Qualifiers *pq, Type *p)
    : Type(sl, pq), pApply(p)
{
//  TRACE2("Created DerivedType(%d, %p)\n", q, p);
}

DerivedType::DerivedType(const SourceLine &sl, Type *p)
    : Type(sl), pApply(p)
{
//  TRACE2("Created DerivedType(%d, %p)\n", q, p);
}

DerivedType::DerivedType(const DerivedType &t)
    : Type(t)
{
    if (t.pApply)
        pApply = t.pApply->Copy();
```

```
}

DerivedType:: DerivedType()
{
    delete pApply;
}

void DerivedType::Render(SourceLine &sl, MString &str) const
{
    if (pApply)
        pApply->Render(sl, str);
}

void DerivedType::Mangle(MString &str) const
{
    if (pApply)
        pApply->Mangle(str);
}

#ifdef DEBUG
void DerivedType::Dump() const
{
    // Called by base classes only.
    if (pApply)
        pApply->Dump();
}
#endif

DerivedType *DerivedType::GetDerivedType()
{
    return this;
}

Type *DerivedType::ReplacePlaceholder(Type *p)
{
    if (pApply)
        pApply = pApply->ReplacePlaceholder(p);
    else
    {
//      TRACE0("Doing Placeholder replace in DerivedType\n");
        pApply = p;
    }
    return this;
}

Type *DerivedType::GetNext()
{
    return pApply;
}

//
// class PointerType : DerivedType
//

PointerType::PointerType(const SourceLine &sl, Qualifiers *pq, Type *p)
    : DerivedType(sl, pq, p)
{
}

PointerType::PointerType(const SourceLine &sl, Type *p)
    : DerivedType(sl, p)
{
}

PointerType::PointerType(const PointerType &t)
    : DerivedType(t)
```

```
{
}

PointerType:: PointerType()
{
}

PointerType *PointerType::Copy() const
{
    return new PointerType(*this);
}

void PointerType::Render(SourceLine &sl, MString &str) const
{
//  TRACE0("Rendering pointer type\n");
    MString str2 = LineCorrect(sl);

    str2 += "(*";
    str2 += GetQualifiers(sl);
    str2 += str;
    str2 += ')';
    str = str2;
    DerivedType::Render(sl, str);
}

void PointerType::Mangle(MString &str) const
{
    Type::Mangle(str);
    str += 'P';
    DerivedType::Mangle(str);
}

#ifdef DEBUG
void PointerType::Dump() const
{
    TRACE1("DUMP: [%p] Pointer Type\n", this);
    DerivedType::Dump();
}
#endif

//
// class FunctionType : DerivedType
//

FunctionType::FunctionType(const SourceLine &sl, Qualifiers *pq, Type *pt,
                           DeclListToken *r)
    : DerivedType(sl, pq, pt), pArgs(r)
{
}

FunctionType::FunctionType(const SourceLine &sl, Type *p, DeclListToken *r)
    : DerivedType(sl, p), pArgs(r)
{
}

FunctionType::FunctionType(const FunctionType &t)
    : DerivedType(t)
{
    if (t.pArgs)
        pArgs = t.pArgs->Copy();
    else
        pArgs = NULL;
}

FunctionType:: FunctionType()
{
```

```
    delete pArgs;
}

FunctionType *FunctionType::Copy() const
{
    return new FunctionType(*this);
}

void FunctionType::Render(SourceLine &sl, MString &str) const
{
//  TRACE0("Rendering function type\n");
    MString pre = LineCorrect(sl);

    pre += GetQualifiers(sl);
    str = pre + str;

    if (pArgs)
    {
        str += "(";
        str += pArgs->GetText(sl);
        str += ")";
    }
    else
        str += "()";
    DerivedType::Render(sl, str);
}

void FunctionType::Mangle(MString &str) const
{
    Type::Mangle(str);
    str += 'F';
    DerivedType::Mangle(str);
}

#ifdef DEBUG
void FunctionType::Dump() const
{
    TRACE1("DUMP: [%p] Function Type\n", this);
    DerivedType::Dump();
}
#endif

FunctionType *FunctionType::GetFunctionType()
{
    return this;
}

DeclListToken *FunctionType::CopyParameterList() const
{
    return pArgs->Copy();
}

Type *FunctionType::CopyReturnType()
{
    return DerivedType::GetNext()->Copy();
}

void FunctionType::MangleReturnType(MString &str)
{
    DerivedType::GetNext()->Mangle(str);
}

unsigned int FunctionType::GetParameterCount() const
{
    return pArgs->Count();
}
```

```
//
// class ArrayType : DerivedType
//

ArrayType::ArrayType(const SourceLine &sl, Qualifiers *pq, Type *pt, Token *r)
    : DerivedType(sl, pq, pt), pBoundExpression(r)
{
}

ArrayType::ArrayType(const SourceLine &sl, Type *p, Token *r)
    : DerivedType(sl, p), pBoundExpression(r)
{
}

ArrayType::ArrayType(const ArrayType &t)
    : DerivedType(t)
{
    if (t.pBoundExpression)
        pBoundExpression = t.pBoundExpression->Copy();
    else
        pBoundExpression = NULL;
}

ArrayType:: ArrayType()
{
    delete pBoundExpression;
}

ArrayType *ArrayType::Copy() const
{
    return new ArrayType(*this);
}

void ArrayType::Render(SourceLine &sl, MString &str) const
{
    MString pre = LineCorrect(sl);

    pre += GetQualifiers(sl);

    str = pre + str;
    str += ’[’;
    if (pBoundExpression)
        str += pBoundExpression->GetText(sl);
    str += ’]’;
    DerivedType::Render(sl, str);
}

void ArrayType::Mangle(MString &str) const
{
    Type::Mangle(str);
    str += ’A’;
    DerivedType::Mangle(str);
}

#ifdef DEBUG
void ArrayType::Dump() const
{
    TRACE1("DUMP: [%p] Array Type\n", this);
    DerivedType::Dump();
}
#endif

//
// class PlaceholderType : Type
//
```

```
PlaceholderType::PlaceholderType(const SourceLine &sl)
    : Type(sl)
{
//  TRACE1("Created PlaceholderType at %p\n", this);
}

PlaceholderType::PlaceholderType(const PlaceholderType &t)
    : Type(t)
{
//  TRACE1("Copied PlaceholderType at %p\n", this);
}

PlaceholderType *PlaceholderType::Copy() const
{
    return new PlaceholderType(*this);
}

void PlaceholderType::Render(SourceLine &sl, MString &str) const
{
    str += LineCorrect(sl);

    str += GetQualifiers(sl);
    str += "/* Placeholder */";
}

void PlaceholderType::Mangle(MString &str) const
{
    Type::Mangle(str);
}

#ifdef DEBUG
void PlaceholderType::Dump() const
{
    TRACE1("DUMP: [%p] Placeholder type\n", this);
}
#endif

Type *PlaceholderType::ReplacePlaceholder(Type *p)
{
//  TRACE0("Doing Placeholder replace in placeholder type.\n");
    delete this;
    return p;
}

//
// class ProcessType : Type
//

ProcessType::ProcessType(const SourceLine &sl, Type *p)
    : Type(sl)
{
//  TRACE1("Created ProcessType at %p\n", this);
    pInnerType = p;
}

ProcessType::ProcessType(const ProcessType &t)
    : Type(t), pInnerType(NULL)
{
    pInnerType = t.pInnerType->Copy();
//  TRACE1("Copied ProcessType at %p\n", this);
}

ProcessType:: ProcessType()
{
    delete pInnerType;
```

```
}

ProcessType *ProcessType::Copy() const
{
    return new ProcessType(*this);
}

void ProcessType::Render(SourceLine &sl, MString &str) const
{
    MString str2 = LineCorrect(sl);
    str2 += GetQualifiers(sl);

    if (pInnerType)
    {
        MString s = "_dcv_";
        pInnerType->Mangle(s);
        str2 += "struct ";
        str2 += s;
        str2 += " *(";
    }
    else
        str2 += "/* Process */";
    str += ')';
    str = str2 + str;
}

void ProcessType::Mangle(MString &str) const
{
    str += '$';
    Type::Mangle(str);
}

#ifdef DEBUG
void ProcessType::Dump() const
{
    TRACE1("DUMP: [%p] Process type\n", this);
    if (pInnerType)
        pInnerType->Dump();
}
#endif
```

B.7.7 dc2/storage.C

```
#include "project.h"
#include "dc2.h"
#include "storage.h"

#define new DEBUG_NEW

StorageClass::StorageClass(const SourceLine &sl, int n)
    : line(sl), nStorage(n)
{
    ASSERT(n <= MAX_CLASS);
}

StorageClass::StorageClass(const StorageClass &t)
    : line(t.line), nStorage(t.nStorage)
{
}

StorageClass *StorageClass::Copy()
{
    return new StorageClass(*this);
```

```
}

void StorageClass::Set(int n)
{
    ASSERT(n <= MAX_CLASS);
    nStorage = n;
}

int StorageClass::Get() const
{
    return nStorage;
}

void StorageClass::ComposeSelf(StorageClass *psc)
{
    if (nStorage & psc->nStorage)
        Compiler::Instance()->Error("Duplicate storage class specifiers");

    nStorage |= psc->nStorage;
    ASSERT(nStorage <= MAX_CLASS);

    if ((nStorage & S_REMOTE) && (nStorage &  (S_EXTERN | S_REMOTE)))
        Compiler::Instance()->Error("Attempt to combine illegal storage class "
                                    "specifiers with 'remote'");

    // Now handle the line numbers.
    if (nStorage == 0 || line.GetLine() > psc->line.GetLine())
        line.Set(psc->line);
}

bool StorageClass::IsClass(int n) const
{
    return (nStorage & n) > 0;
}

const char *StorageClass::szClassNames[] =
{
    "typedef ",
    "extern ",
    "/*remote*/ ",
    "static ",
    "__inline__ ",
    "auto ",
    "register ",
};

void StorageClass::Render(SourceLine &sl, MString &str) const
{
    str += line.LineCorrect(sl);
    sl.Set(line);

    unsigned int n = nStorage;
    ASSERT(n < MAX_CLASS);
    for(int i = 0; n; i++, n >>= 1)
        if (n & 1)
            str += szClassNames[i];
}

const SourceLine &StorageClass::GetSourceLine() const
{
    return line;
}
```

B.7.8 dc2/compose.C

```
#include "project.h"
#include "dc2.h"
#include "type.h"

//
// BasicType Composition
//

Type *BasicType::Compose(Type *p)
{
    // Should be a dynamic_cast really but g++ doesn't cope with them very
    // well, so we cheat instead.

//  TRACE1("BasicType::ComposeSelf called from %d\n", n);

    BasicType *pb = p->GetBasicType();

    if (pb)
        InternalComposeSelf(pb);
    else
        Error("Illegal attempt to compose with basic type");

    // Now we've dealt with it throw it away.
//  TRACE1("Throwing away after compose: %p\n", p);
    delete p;

    return this;
}

void BasicType::InternalComposeSelf(BasicType *pb)
{
    enum { A_SHORT, A_LONG, NUMBER_OF_AS };

    static int nPromotions[NUMBER_OF_TYPES][NUMBER_OF_AS] =
    {
        { T_SHORT, T_LONG }, /* None */
        { 0, 0 }, /* Void */
        { 0, 0 }, /* Char */
        { 0, 0 }, /* Short */
        { T_SHORT, T_LONG }, /* Int */
        { 0, T_LLONG }, /* Long */
        { 0, 0 }, /* LLong */
        { 0, T_DOUBLE }, /* Float */
        { 0, T_LDOUBLE }, /* Double */
        { 0, 0 }, /* LDouble */
    };

    // First check there are no problems with combining the signs

    if ((this->nSign != T_NOSIGN) && (pb->nSign != T_NOSIGN))
        Error("Attempt to combine multiple unsigned/signed modifiers");
    else
        this->nSign |= pb->nSign;

    ASSERT(this->nSimpleType < NUMBER_OF_TYPES);
    ASSERT(pb->nSimpleType < NUMBER_OF_TYPES);

    bool bSuccess = false;
    // Easy cases, if one of them has nothing in it.
    if (this->nSimpleType == T_NONE)
    {
        nSimpleType = pb->nSimpleType;
        bSuccess = true;
    }
    else if (pb->nSimpleType == T_NONE)
```

```
    {
        bSuccess = true;
    }
    else if (this->nSimpleType == T_SHORT)
    {
        if (nPromotions[pb->nSimpleType][A_SHORT])
        {
            nSimpleType = nPromotions[pb->nSimpleType][A_SHORT];
            bSuccess = true;
        }
    }
    else if (this->nSimpleType == T_LONG)
    {
        if (nPromotions[pb->nSimpleType][A_LONG])
        {
            nSimpleType = nPromotions[pb->nSimpleType][A_LONG];
            bSuccess = true;
        }

    }
    else if (pb->nSimpleType == T_SHORT)
    {
        if (nPromotions[this->nSimpleType][A_SHORT])
        {
            nSimpleType = nPromotions[this->nSimpleType][A_SHORT];
            bSuccess = true;
        }
    }
    else if (pb->nSimpleType == T_LONG)
    {
        if (nPromotions[this->nSimpleType][A_LONG])
        {
            nSimpleType = nPromotions[this->nSimpleType][A_LONG];
            bSuccess = true;
        }

    }

    if (!bSuccess)
        Compiler::Instance()->Error("Attempt to combine incompatible types");

    ComposeQualifiers(pb);
}

//
// CompoundType composition
//

Type *CompoundType::Compose(Type *p)
{
//  TRACE1("CompoundType::ComposeSelf called from %d\n", n);
    Error("Illegal attempt to compose with struct/union/enum type.");

    delete p;

    return this;
}

//
// UserType composition
//

Type *UserType::Compose(Type *p)
{
//  TRACE1("UserType::ComposeSelf called from %d\n", n);
    Error("Illegal attempt to compose with typedef type.");
```

```
    delete p;

    return this;
}

//
// DerivedType composition
//

Type *DerivedType::Compose(Type *p)
{
//  TRACE2("DerivedType::ComposeSelf called from %d on %p\n", n, p);
    // OK, if we've already got one apply it to the next.

    if (pApply)
        pApply = pApply->Compose(p);
    else
        InternalComposeSelf(p);

    return this;
}

void DerivedType::InternalComposeSelf(Type *p)
{
    ASSERT(pApply == NULL);
#ifdef DEBUG
//  TRACE0("@@ I am\n");
//  Dump();
//  TRACE0("@@ Composing with\n");
//  p->Dump();
//  TRACE0("@@\n");
#endif
    pApply = p;
}

Type *PlaceholderType::Compose(Type *p)
{
//  TRACE0("Doing placeholder thang\n");
    delete this;
    return p;
}

Type *ProcessType::Compose(Type *p)
{
    Error("Illegal attempt to compose with process<...> type");
    delete p;
    return this;
}
```

B.7.9 dc2/codegen.C

```
#include "project.h"
#include "dc2.h"
#include "tokid.h"

#define new DEBUG_NEW

DeclListToken *DeclToken::BuildRemoteFunctionDeclaration()
{
    // Use S_BETWEEN since the last decl is a function and doesn't need the ';'
    DeclListToken *pReturn = new DeclListToken(DECLLIST, ';',
                                               DeclListToken::S_ASK);
```

```
    FunctionType *pf = pType->GetFunctionType();
    if (pf)
    {
        if (!Compiler::Instance()->IsSymbol(NS_REMOTEFN, strIdentifier))
        {
            Compiler::Instance()->AddSymbol(NS_REMOTEFN, strIdentifier);

            // First write out the parameter structure
            MString s = "_dcp_";
            s += strIdentifier;
            DeclListToken *pParams = pf->CopyParameterList();

            pParams->NumberIdentifiers();
            pReturn->AdoptRight(DeclToken::GenerateStructure(line, s,
                                                             "_dch_invoke",
                                                             pParams));

            // Now write out the inline launching function
            pReturn->AdoptRight(BuildRemotePackFunction(pf));
        }
    }
    else
        Error("'remote' can only be applied to functions");
    return pReturn;
}

DeclListToken *DeclToken::BuildRemoteFunctionDefinition()
{
    DeclListToken *pReturn = new DeclListToken(DECLLIST, ';',
                                               DeclListToken::S_ASK);

    FunctionType *pf = pType->GetFunctionType();
    if (pf)
    {
        pReturn->AdoptRight(BuildRemoteUnpackFunction(pf));
    }
    return pReturn;
}

#if 0
Token *DeclToken::BuildPayloadCalc(const char *v)
{
    MString strVar = v;
    CompoundToken *p = new CompoundToken(STMT);
    p->AdoptRight(new TextToken(IDENTIFIER, strVar + ".hdr.cbPayload"));
    p->AdoptRight(new CharToken('='));
    p->AdoptRight(new TextToken(GENTOK, "sizeof("));
    p->AdoptRight(new TextToken(IDENTIFIER, strVar));
    p->AdoptRight(new TextToken(GENTOK, ") - sizeof("));
    p->AdoptRight(new TextToken(IDENTIFIER, strVar + ".hdr"));
    p->AdoptRight(new TextToken(GENTOK, ");"));
    return p;
}
#endif

DeclToken *DeclToken::BuildRemotePackFunction(FunctionType *pf)
{
    DeclListToken *pParams = pf->CopyParameterList();
    pParams->NumberIdentifiers();
    unsigned int nArgs = pParams->Count();

    MString strVarType = "_dcv_";
    pf->MangleReturnType(strVarType);

    Type *pReturnType =
        new PointerType(line, new CompoundType(line, CompoundType::C_STRUCT,
```

```
                                           strVarType, NULL, true));
    MString strPackFunction = "_dcf_";
    strPackFunction += strIdentifier;
    StorageClass *psc;
    psc = new StorageClass(line,
                           StorageClass::S_INLINE | StorageClass::S_STATIC);
//  Type *pt = new BasicType(BasicType::T_INT, BasicType::T_NOSIGN);
    DeclToken *pFunction;
    pFunction = new DeclToken(line, FUNC, psc,
                              new FunctionType(line, pReturnType, pParams),
                              strPackFunction);

    MString strOutParams = "_dct_p";
    MString strRetVal = "_dct_r";
    // Now build up the implementation
    CompoundToken *pStmts = new CompoundToken(STMTLIST);

    // Declare the variables
    DeclListToken *pDecls = new DeclListToken(DECLLIST, ';',
                                              DeclListToken::S_POST);
//  Type *pt;
    MString strParamStruct = "_dcp_";
    strParamStruct += strIdentifier;

    pDecls->AdoptRight(BuildPackDecl(line, strParamStruct, strOutParams,
                                     "_dcx_allocargs"));
    pDecls->AdoptRight(BuildPackDecl(line, strVarType, strRetVal,
                                     "_dcx_allocvar"));
    pStmts->AdoptRight(pDecls);

    char szName[7] = "->p_aa";
    for(unsigned int i = 0; i < nArgs; i++)
    {
        CompoundToken *p = new CompoundToken(STMT);
        p->AdoptRight(new TextToken(line, IDENTIFIER, strOutParams + szName));
        p->AdoptRight(new CharToken(line, '='));
        p->AdoptRight(new TextToken(line, IDENTIFIER, szName + 2));
        p->AdoptRight(new CharToken(line, ';'));
        pStmts->AdoptRight(p);

        if (szName[5] == 'z')
            szName[4]++, szName[5] = 'a';
        else
            szName[5]++;
    }

    MString strUnpackFunction = "\"_dcu_";
    strUnpackFunction += strIdentifier;
    strUnpackFunction += "\"";

    CompoundToken *p;
    p = new CompoundToken(STMT);
    p->AdoptRight(new TextToken(line, GENTOK,
                                "_dcx_setfn((struct _dch_invoke *)"));
    p->AdoptRight(new TextToken(line, IDENTIFIER, strOutParams));
    p->AdoptRight(new CharToken(line, ','));
    p->AdoptRight(new TextToken(line, IDENTIFIER, strUnpackFunction));
    p->AdoptRight(new TextToken(line, GENTOK, ");"));

    pStmts->AdoptRight(p);

    p = new CompoundToken(STMT);
    p->AdoptRight(new TextToken(line, GENTOK, "_dcx_exec((char *)0, "));
    p->AdoptRight(new TextToken(line, GENTOK, "(struct _dch_var *)"));
    p->AdoptRight(new TextToken(line, IDENTIFIER, strRetVal));
    p->AdoptRight(new CharToken(line, ','));
```

```
    p->AdoptRight(new TextToken(line, GENTOK, "(struct _dch_invoke *)"));
    p->AdoptRight(new TextToken(line, IDENTIFIER, strOutParams));
    p->AdoptRight(new TextToken(line, GENTOK, ");"));
    pStmts->AdoptRight(p);

    p = new CompoundToken(STMT);
    p->AdoptRight(new TextToken(line, RETURN, "return"));
    p->AdoptRight(new TextToken(line, IDENTIFIER, strRetVal));
    p->AdoptRight(new CharToken(line, ';'));
    pStmts->AdoptRight(p);

    CompoundToken *pImplementation = new CompoundToken(STMT);
    pImplementation->AdoptRight(new CharToken(line, '{'));
    pImplementation->AdoptRight(pStmts);
    pImplementation->AdoptRight(new CharToken(line, '}'));

    pFunction->SetImplementation(pImplementation);

    return pFunction;
}

DeclToken *DeclToken::BuildRemoteUnpackFunction(FunctionType *pf)
{
    //DeclListToken *pParams = pf->CopyParameterList();
    //pParams->NumberIdentifiers();
    //unsigned int nArgs = pParams->Count();

    // Start by building up the function parameter list.

    DeclListToken *pInParams = new DeclListToken(PARMLIST, ',',
                                                 DeclListToken::S_BETWEEN);
    Type *pParmType;
    // The first parameter is a struct _dch_invoke *
    pParmType = new PointerType(line, new CompoundType(line,
                                                        CompoundType::C_STRUCT,
                                                        "_dch_invoke", NULL,
                                                        true));
    pInParams->AdoptRight(DeclToken::SwallowType(pParmType, "_dct_pi"));

    // The second parameter is a struct _dch_var *

    pParmType = new PointerType(line, new CompoundType(line,
                                                        CompoundType::C_STRUCT,
                                                        "_dch_var", NULL,
                                                        true));
    pInParams->AdoptRight(DeclToken::SwallowType(pParmType, "_dct_pv"));

    // Now build up the function header

    MString strUnpackFunction = "_dcu_";
    strUnpackFunction += strIdentifier;
    Type *pt = new BasicType(line, BasicType::T_INT, BasicType::T_NOSIGN);
    DeclToken *pFunction = new DeclToken(line, FUNC, new StorageClass(line),
                                         new FunctionType(line, pt, pInParams),
                                         strUnpackFunction);

    // Now build up the implementation
    MString strValueType = "_dcv_";
    pf->MangleReturnType(strValueType);

    MString strParamType = "_dcp_";
    strParamType += strIdentifier;

    CompoundToken *pStmts = new CompoundToken(STMTLIST);

    MString strParamVar = "_dct_pp";
```

```
    MString strRetVar = "_dct_pr";

    pStmts->AdoptRight(DeclToken::BuildInitialization(line, strParamType,
                                                      strParamVar, "_dct_pi"));
    pStmts->AdoptRight(DeclToken::BuildInitialization(line, strValueType,
                                                      strRetVar, "_dct_pv"));

    // Now build up the parameters for the function

    CompoundToken *pArgs = new CompoundToken(PARMLIST);
    char szName[7] = "->p_aa";
    for(unsigned int i = 0; i < pf->GetParameterCount(); i++)
    {
        if (i > 0)
            pArgs->AdoptRight(new CharToken(line, ','));

        pArgs->AdoptRight(new TextToken(line, IDENTIFIER,
                                        strParamVar + szName));

        if (szName[5] == 'z')
            szName[4]++, szName[5] = 'a';
        else
            szName[5]++;
    }

    CompoundToken *pExecute = new CompoundToken(STMTLIST);
    pExecute->AdoptRight(new TextToken(line, IDENTIFIER, strRetVar));
    pExecute->AdoptRight(new TextToken(line, PTR_OP, "->"));
    pExecute->AdoptRight(new TextToken(line, IDENTIFIER, "val"));
    pExecute->AdoptRight(new CharToken(line, '='));
    pExecute->AdoptRight(new TextToken(line, IDENTIFIER, strIdentifier));
    pExecute->AdoptRight(new CharToken(line, '('));
    pExecute->AdoptRight(pArgs);
    pExecute->AdoptRight(new CharToken(line, ')'));
    pExecute->AdoptRight(new CharToken(line, ';'));
    pExecute->AdoptRight(new TextToken(line, RETURN, "return"));
    pExecute->AdoptRight(new TextToken(line, CONSTANT, "0"));
    pExecute->AdoptRight(new CharToken(line, ';'));
    pStmts->AdoptRight(pExecute);

    CompoundToken *pImplementation = new CompoundToken(STMT);
    pImplementation->AdoptRight(new CharToken(line, '{'));
    pImplementation->AdoptRight(pStmts);
    pImplementation->AdoptRight(new CharToken(line, '}'));

    pFunction->SetImplementation(pImplementation);

    return pFunction;
}

DeclListToken *DeclToken::HandleDeclaration()
{
    DeclListToken *pNew = new DeclListToken(DECLLIST, ';',
                                            DeclListToken::S_ASK);
    pNew->AdoptRight(this);

    if (pStorage->IsClass(StorageClass::S_REMOTE))
    {
        //TRACE1("WE'RE REMOTE! %s\n", (const char *)GetText());

        pNew->ConcatRight(BuildRemoteFunctionDeclaration());
    }
    else
    {
        pNew->ConcatRight(BuildVarStructs());
    }
```

```
    return pNew;
}

DeclListToken *DeclToken::BuildVarStructs()
{
    DeclListToken *pReturn = new DeclListToken(DECLLIST, ';',
                                              DeclListToken::S_ASK);

    if (pStorage->IsClass(StorageClass::S_TYPEDEF))
    {
        DeclToken *pd = new DeclToken(line, DECLSPEC, new StorageClass(line),
                                      new UserType(line, strIdentifier), "");
        pReturn->AdoptRight(BuildVarStruct(line, pd));
    }

    if (pType)
    {
        Type *t = pType->GetDefinitionType();
        if (t)
        {
            DeclToken *p;
            p = BuildVarStruct(line, new DeclToken(line, DECLSPEC,
                                                   new StorageClass(line),
                                                   t, ""));
            pReturn->AdoptRight(p);
        }
    }
    return pReturn;
}

DeclToken *DeclToken::BuildVarStruct(const SourceLine &sl, DeclToken *pDefn)
{
    // Should put something in here to stop it doing the ones
    // that were generated by process<type> declarations.
    // At the moment it ends up generating structures like:
    // _dcv_G6_dcv_i, but these don't actually do any harm.

    ASSERT(pDefn);
    ASSERT(pDefn->pType);

    pDefn->ChangeIdentifier("val");

    MString s = "_dcv_";
    pDefn->pType->Mangle(s);

    if (Compiler::Instance()->IsSymbol(NS_VARGEN, s))
    {
        delete pDefn;
        return NULL;
    }
    else
    {
        Compiler::Instance()->AddSymbol(NS_VARGEN, s);
        return DeclToken::GenerateStructure(sl, s, "_dch_var", pDefn);
    }
}

DeclListToken *DeclToken::HandleDefinition()
{
    DeclListToken *pNew = new DeclListToken(DECLLIST, ';',
                                           DeclListToken::S_ASK);

    if (pStorage->IsClass(StorageClass::S_REMOTE))
    {
        //TRACE1("WE'RE REMOTE! %s\n", (const char *)GetText());
```

```
        pNew->ConcatRight(BuildRemoteFunctionDeclaration());
        pNew->AdoptRight(this);
        pNew->ConcatRight(BuildRemoteFunctionDefinition());
    }
    else
        pNew->AdoptRight(this);
    return pNew;
}

DeclListToken *DeclListToken::HandleDeclarations()
{
    DeclListToken *pNew = new DeclListToken(DECLLIST, ';',
                                        DeclListToken::S_ASK);

    TokenList::iterator pCurrent, pNext;
    pNext = listChildren.begin();

    while (pNext != listChildren.end())
    {
        pCurrent = pNext;
        ++pNext;

        pNew->ConcatRight((*pCurrent)->HandleDeclaration());

        listChildren.erase(pCurrent);
    }
    delete this;
    return pNew;
}

void DeclListToken::NumberIdentifiers()
{
    char szName[5] = "p_aa";

    for(TokenList::iterator i = listChildren.begin();
        i != listChildren.end(); ++i)
    {
        (*i)->ChangeIdentifier(szName);
        if (szName[3] == 'z')
            szName[2]++, szName[3] = 'a';
        else
            szName[3]++;
    }
}

DeclToken *DeclToken::BuildInitialization(const SourceLine &sl, const char *t,
                                          const char *v, const char *i)
{
    Type *pType = new PointerType(sl, new CompoundType(sl,
                                                CompoundType::C_STRUCT,
                                                t, NULL, true));
    DeclToken *pToken = DeclToken::SwallowType(pType, v);

    MString strInitializer = "(struct ";
    strInitializer += t;
    strInitializer += '*';
    strInitializer += ')';
    strInitializer += i;
    strInitializer += ';';
    pToken->SetInitializer(new TextToken(sl, GENTOK, strInitializer));
    return pToken;
}

DeclToken *DeclToken::GenerateStructure(const SourceLine &sl, const char *s,
                                        const char *h, DeclListToken *t)
{
```

```
    t->SetSeparator(';', DeclListToken::S_POST);

    // First generate the header structure reference.

    DeclToken *pHeader = SwallowType(new CompoundType(sl,
                                                      CompoundType::C_STRUCT,
                                                      h, NULL, true));
    pHeader->ChangeIdentifier("hdr");
    t->AdoptLeft(pHeader);

    CompoundType *pType = new CompoundType(sl, CompoundType::C_STRUCT,
                                           s, t, true);

    return SwallowType(pType);
}

DeclToken *DeclToken::GenerateStructure(const SourceLine &sl, const char *s,
                                        const char *h, DeclToken *t)
{
    DeclListToken *p = DeclListToken::SwallowDeclToken(t, ';',
                                                       DeclListToken::S_POST);
    return GenerateStructure(sl, s, h, p);
}

Type *Type::HandleProcessType()
{
    // Someone wants a process type based on this type. What shall we do?
    MString s = "_dcv_";
    Mangle(s);
//  TRACE1("Mangled: %s\n", (const char *)s);
    delete this;
    return new PointerType(line, new CompoundType(line, CompoundType::C_STRUCT,
                                                  s, NULL, true));
}

#ifdef PREAMBLE
DeclListToken *DeclToken::GeneratePreambleDecls()
{
    DeclListToken *pList = new DeclListToken(DECLLIST, ';',
                                             DeclListToken::S_POST);
    BasicType::BuildSimpleStructs(pList);
    return pList;
}

void BasicType::BuildSimpleStructs(DeclListToken *l)
{

    BasicType t(0, 0);

    for(int b = 1; b < NUMBER_OF_TYPES; b++)
    {
        for(int s = 0; s < NUMBER_OF_SIGNS; s++)
        {
            t.nSimpleType = b;
            t.nSign = s;
            DeclToken *pDecl = DeclToken::SwallowType(t.Copy());
            l->AdoptRight(DeclToken::BuildVarStruct(pDecl));
        }
    }
}

#endif

#if 0
Token *Compiler::GenerateRemoteCall(const SourceLine &sl, TextToken *pName,
                                    Token *pArgs)
```

```
{
    CompoundToken *c = new CompoundToken(EXPR);
    MString s = "_dcf_";
    s += pName->GetLiteralText();

    c->AdoptRight(new TextToken(sl, IDENTIFIER, s));
    c->AdoptRight(new CharToken(sl, '('));
    c->AdoptRight(pArgs);
    c->AdoptRight(new CharToken(sl, ')'));

    delete pName;
    return c;
}
#endif

Token *Compiler::GenerateRemoteWait(const SourceLine &sl, Token *pVar)
{
    CompoundToken *c = new CompoundToken(EXPR);
    c->AdoptRight(new TextToken(sl, GENTOK, "({"));
    c->AdoptRight(new TextToken(sl, GENTOK, "typeof("));
    c->AdoptRight(pVar);
    c->AdoptRight(new TextToken(sl, GENTOK, ") _dct= ("));
    c->AdoptRight(pVar->Copy());
    c->AdoptRight(new TextToken(sl, GENTOK, ");"));
    c->AdoptRight(new TextToken(sl, IDENTIFIER, "_dcx_wait"));
    c->AdoptRight(new CharToken(sl, '('));
    c->AdoptRight(new TextToken(sl, GENTOK, "(struct _dch_var *)_dct);"));
    c->AdoptRight(new TextToken(sl, GENTOK, "_dct->val;"));
    c->AdoptRight(new TextToken(sl, GENTOK, "})"));
    return c;
}

Token *Compiler::GenerateRemoteSelect(const SourceLine &sl,
                                      BoundedArrayListToken *p)
{
    CompoundToken *c = new CompoundToken(EXPR);
    c->AdoptRight(new TextToken(sl, IDENTIFIER, "_dcx_select"));
    c->AdoptRight(new CharToken(sl, '('));
    c->AdoptRight(p);
    c->AdoptRight(new CharToken(sl, ')'));
    return c;
}

DeclToken *DeclToken::BuildPackDecl(const SourceLine &sl, const char *szStruct,
                                    const char *szName, const char *szFunc)
{
    Type *pType;
    CompoundToken *pInitializer = new CompoundToken(GENTOK);
    pType = new CompoundType(sl, CompoundType::C_STRUCT,
                             szStruct, NULL, true);

    pInitializer->AdoptRight(new TextToken(sl, IDENTIFIER, szFunc));
    pInitializer->AdoptRight(new CharToken(sl, '('));
    pInitializer->AdoptRight(new TextToken(sl, SIZEOF, "sizeof"));
    pInitializer->AdoptRight(new CharToken(sl, '('));
    pInitializer->AdoptRight(DeclToken::SwallowType(pType->Copy()));
    pInitializer->AdoptRight(new CharToken(sl, ')'));
    pInitializer->AdoptRight(new CharToken(sl, ')'));

    DeclToken *p = new DeclToken(sl, DECL, new StorageClass(sl),
                                 new PointerType(sl, pType), szName);
    p->SetInitializer(pInitializer);
    return p;
}

#if 0
```

```
Token *Compiler::GenerateDistFor(const SourceLine &sl,
                                 Token *pExp1, Token *pExp2, Token *pExp3,
                                 Token *pAssign, TextToken *pFunc,
                                 Token *pArgs)
{

    CompoundToken *pCode = new CompoundToken(STMT);
    pCode->AdoptRight(new CharToken(sl, '{'));

    // Bucket variable declaration
    {
        CompoundToken *pStmt = new CompoundToken(STMT);
        DeclToken *pBucketDecl;
        Type *pBucketType;
        StorageClass *pBucketStorage = new StorageClass(sl);
        pBucketType = new UserType(sl, new Qualifiers(sl), "_dct_bucket");
        pBucketDecl = new DeclToken(sl, DECLSPEC, pBucketStorage, pBucketType,
                                    "_dct_b");
        {
            CompoundToken *pInit = new CompoundToken(EXPR);
            pInit->AdoptRight(new TextToken(sl, GENTOK, "_dcx_bucket_create"));
            pInit->AdoptRight(new CharToken(sl, '('));
//          pInit->AdoptRight(new TextToken(sl, SIZEOF, "sizeof"));
//          pInit->AdoptRight(new CharToken(sl, '('));
//          pInit->AdoptRight(pAssign->Copy());
//          pInit->AdoptRight(new CharToken(sl, ')'));
            pInit->AdoptRight(new CharToken(sl, ')'));

            pBucketDecl->SetInitializer(pInit);
        }
        pStmt->AdoptRight(pBucketDecl);
        pStmt->AdoptRight(new CharToken(sl, ';'));

        pCode->AdoptRight(pStmt);
    }

    // Now the for loop
    {
        CompoundToken *pLoop = new CompoundToken(STMT);
        pLoop->AdoptRight(new TextToken(sl, FOR, "for"));
        pLoop->AdoptRight(new CharToken(sl, '('));
        pLoop->AdoptRight(pExp1);
        pLoop->AdoptRight(pExp2);
        pLoop->AdoptRight(pExp3);
        pLoop->AdoptRight(new CharToken(sl, ')'));
        pLoop->AdoptRight(new TextToken(sl, IDENTIFIER, "_dcx_bucket_add"));
        pLoop->AdoptRight(new CharToken(sl, '('));
        pLoop->AdoptRight(new TextToken(sl, IDENTIFIER, "_dct_b"));
        pLoop->AdoptRight(new CharToken(sl, ','));
        pLoop->AdoptRight(new CharToken(sl, '&'));
        pLoop->AdoptRight(new CharToken(sl, '('));
        pLoop->AdoptRight(pAssign);
        pLoop->AdoptRight(new CharToken(sl, ')'));
        pLoop->AdoptRight(new CharToken(sl, ','));
        pLoop->AdoptRight(new TextToken(sl, GENTOK, "(struct _dch_var *)"));
        pLoop->AdoptRight(Compiler::GenerateRemoteCall(sl, pFunc, pArgs));
        pLoop->AdoptRight(new CharToken(sl, ')'));
        pLoop->AdoptRight(new CharToken(sl, ';'));

        pCode->AdoptRight(pLoop);
    }

    // Now the wait function
    {
        CompoundToken *pWait = new CompoundToken(STMT);
        pWait->AdoptRight(new TextToken(sl, IDENTIFIER, "_dcx_bucket_wait"));
```

```
        pWait->AdoptRight(new CharToken(sl, '('));
        pWait->AdoptRight(new TextToken(sl, IDENTIFIER, "_dct_b"));
        pWait->AdoptRight(new CharToken(sl, ')'));
        pWait->AdoptRight(new CharToken(sl, ';'));

        pCode->AdoptRight(pWait);
    }
    pCode->AdoptRight(new CharToken(sl, '}'));
    return pCode;
}
#endif
```

B.7.10 dc2/qual.C

```
#include "project.h"
#include "dc2.h"
#include "qual.h"

Qualifiers::Qualifiers(const SourceLine &sl, int n)
    : line(sl)
{
    nQualifiers = n;
}

Qualifiers::Qualifiers(const Qualifiers &q)
    : line(q.line), nQualifiers(q.nQualifiers)
{
}

Qualifiers *Qualifiers::Copy()
{
    return new Qualifiers(*this);
}

MString Qualifiers::GetText(SourceLine &sl) const
{
    MString str = line.LineCorrect(sl);
    sl.Set(line);

    if (nQualifiers & Q_CONST)
        str += "const ";
    if (nQualifiers & Q_VOLATILE)
        str += "volatile ";
    if (nQualifiers & Q_RESTRICT)
        str += "restrict ";
//  if (nQualifiers == 0)
//      str += "/*noqual*/ ";
    return str;
}

void Qualifiers::Mangle(MString &str) const
{
    if (nQualifiers & Q_CONST)
        str += "C";
    if (nQualifiers & Q_VOLATILE)
        str += "V";
    if (nQualifiers & Q_RESTRICT)
        str += "R";
}

// void Qualifiers::Set(int q)
// {
//      nQualifiers = q;
// }
```

```
// void Qualifiers::Set(const Qualifiers &q)
// {
//      nQualifiers = q.nQualifiers;
// }

void Qualifiers::Merge(int q)
{
    if (nQualifiers & q)
        Compiler::Instance()->Error("Illegal combination of qualifiers");
    nQualifiers |= q;
}

void Qualifiers::Merge(Qualifiers *pq)
{
    Merge(pq->nQualifiers);
}

void Qualifiers::Clear()
{
    nQualifiers = Q_NONE;
}

const SourceLine &Qualifiers::GetSourceLine() const
{
    return line;
}
```

B.7.11 dc2/srcline.C

```
#include "project.h"
#include "dc2.h"
#include "srcline.h"

//SourceLine SourceLine::Null;
SourceLine::MStringSet SourceLine::setFiles;

SourceLine::SourceLine()
{
    nLine = -1;
    ptrFilename = Lookup("");
}

SourceLine::SourceLine(int n, const char *s)
{
    nLine = n;
    ptrFilename = Lookup(s);
}

SourceLine::SourceLine(const SourceLine &t)
{
    Set(t);
}

SourceLine::Pointer SourceLine::Lookup(const char *s)
{
    pair<Pointer, bool> p = setFiles.insert(s);
    return p.first;
}

MString SourceLine::LineCorrect(const SourceLine &sl) const
{
    MString str;
    if ((nLine != sl.nLine) || (*ptrFilename != *(sl.ptrFilename)))
    {
```

```
//      TRACE4("Correcting line from %s:%d to %s:%d\n",
//             (const char *) *(sl.ptrFilename), sl.nLine,
//             (const char *) *ptrFilename, nLine);

        if (*ptrFilename == *(sl.ptrFilename) && (nLine > sl.nLine))
        {
            // We're in the same file, and we just need to advance by the
            // correct number of lines
            int n = nLine - sl.nLine;
            for(int i = 0; i < n; i++)
                str += '\n';
        }
        else
        {
            register size_t nSize = ((sizeof(int) * CHAR_BIT)/3 + 9
                                    + (*ptrFilename).GetLength());
            sprintf(str.Grab(nSize), "\n# %d \"%s\"\n",
                    nLine, (const char *)*ptrFilename);
            str.Release();
        }
    }
    return str;
}

const SourceLine &SourceLine::GetSourceLine() const
{
    return *this;
}

void SourceLine::Advance()
{
    ++nLine;
}

void SourceLine::Set(int n, const char *s)
{
    nLine = n;
    if (s)
        ptrFilename = Lookup(s);
}

void SourceLine::Set(const SourceLine &l)
{
    nLine = l.nLine;
    ptrFilename = l.ptrFilename;
}

SourceLine SourceLine::Null()
{
    return SourceLine();
}

const char *SourceLine::GetFile() const
{
    return *ptrFilename;
}

int SourceLine::GetLine() const
{
    return nLine;
}
```

B.7.12 dc2/dc2.h

```
#ifndef DC2_H
#define DC2_H 1

#include "token.h"
#include "type.h"

#include "compiler.h"

extern int yyparse();

#endif
```

B.7.13 dc2/compiler.h

```
#ifndef COMPILER_H
#define COMPILER_H 1

#include <fstream.h>
#include <objhash.h>
#include "syminfo.h"
#include "lexer.h"
#include "srcline.h"

class Token;
class TextToken;
class BoundedArrayListToken;
class BoundedArrayToken;

class Compiler
{
    SymbolTable symtab;
    yyFlexLexer lexer;

    istream &fileInput;
    ostream &fileOutput;

    MString strPreambleFile;
    MString strInputFile;

    MString strPthreadFile;

    int nError;

    static Compiler *pInstance;

    int nCurrentScope;

    SourceLine CurrentLine;
    SourceLine OutputLine;

protected:
    void OutputPreamble();
    bool FindPreambleFile();
    bool FindPthreadFile();
    void IncludeFile(const char *);

public:
    Compiler(istream &, ostream &, const char *);
     Compiler();

    void Error(const char *s);

    int GetError() { return nError; }
```

```
    static Compiler *Instance();

    bool IsSymbol(Namespace ns, const char *s);
    void AddSymbol(Namespace ns, const char *s);
    void EnterScope();
    void LeaveScope();

    void Count();
    int Lex() { return lexer.yylex(); }
    void Output(const Token *p);

    static Token *GenerateRemoteCall(const SourceLine &sl, TextToken *pName,
                                     Token *pArgs);
    static Token *GenerateRemoteWait(const SourceLine &sl, Token *pVar);
    static Token *GenerateRemoteSelect(const SourceLine &sl,
                                       BoundedArrayListToken *);
    static Token *GenerateDistFor(const SourceLine &sl,
                                  Token *, Token *, Token *, Token *,
                                  TextToken *, Token *);

    static void HandleDirective(const char *s);

    static const SourceLine &GetCurrentLine()
    {
        return Compiler::Instance()->CurrentLine;
    }

};

#define yyerror(A) Compiler::Instance()->Error(A)

inline void Error(const char *s)
{
    Compiler::Instance()->Error(s);
}

#endif
```

B.7.14 dc2/lexer.h

```
#include <FlexLexer.h>
```

B.7.15 dc2/syminfo.h

```
#ifndef SYMINFO_H
#define SYMINFO_H 1

#include <mstring.h>

// Valid namespaces

enum Namespace
{
    NS_GLOBAL, /* Global symbols, just typedefs atm */
    NS_STRUCT, NS_UNION, NS_ENUM, /* Compound type symbols */
    NS_VARGEN, /* Generated _dcv_ structures */
    NS_REMOTEFN  /* Generated remote function structures */
};

class SymbolKey
{
    Namespace ns;
```

```
    const char *key;

public:
    SymbolKey(Namespace n, const char *k)
        : ns(n), key(k) {}
    SymbolKey(const SymbolKey &t)
        : ns(t.ns), key(t.key) {}

    operator ==(const SymbolKey &t) const
    {
        return (t.ns == ns) && !strcmp(t.key, key);
    }
    operator const char *() const
    {
        return key;
    }
};

class SymbolInfo
{
    Namespace ns;
    MString strSymbolName;
    int nScopeLevel;

public:
    SymbolInfo(const SymbolInfo &t)
        : ns(t.ns), strSymbolName(t.strSymbolName), nScopeLevel(t.nScopeLevel) {}
    SymbolInfo(Namespace nm, const char *s, int n)
        : ns(nm), strSymbolName(s), nScopeLevel(n) {}

    const char *GetSymbolName() const { return strSymbolName; }

    bool IsKeyMatch(const SymbolKey &k) const
    {
        return (k == SymbolKey(ns, strSymbolName));
    }

    SymbolKey GetKey() const { return SymbolKey(ns, strSymbolName); }

    int GetScopeLevel() { return nScopeLevel; }

    static unsigned int HashFunction(const SymbolKey &k)
    {
        const char *s = k;

        int n = 0;
        while(*s)
        {
            n += *s++;
            n <<= 1;
        }
        return n;
    }
};

typedef ObjectHash<SymbolInfo, const SymbolKey &> SymbolTable;
typedef ObjectHashIterator<SymbolInfo, const SymbolKey &> SymbolTableIterator;

#endif
```

B.7.16 dc2/token.h

```
#ifndef TOKEN_H
#define TOKEN_H 1
```

```
#include <list>
#include <mstring.h>

#include "storage.h"
#include "qual.h"
#include "srcitem.h"

class Type;
class FunctionType;
class UserType;
class Token;
class DeclToken;
class DeclListToken;

// typedef list<Token *> TokenList;
// typedef list<DeclToken *> DeclTokenList;

class Token : public SourceItem
{
    int nTokenId;

public:
    Token(int);
    Token(const Token &t);
    virtual  Token();
    virtual Token *Copy() const = 0;
    int GetTokenId() const;
    virtual MString GetText(SourceLine &sl) const = 0;

    virtual DeclToken *GetDeclToken();
#ifdef DEBUG
    virtual void DumpTree(int nDepth = 0) const;
#endif
};

template<class T>
class ListToken : public Token
{
protected:
    typedef list<T *> TokenList;
    TokenList listChildren;

public:
    ListToken(int);
    ListToken(const ListToken &);
    virtual  ListToken();
    virtual ListToken *Copy() const = 0;

    void AdoptLeft(T *);
    void AdoptRight(T *);
    void ConcatRight(ListToken<T> *);
    unsigned int Count() const;

    virtual MString GetText(SourceLine &sl) const = 0;

#ifdef DEBUG
    virtual void DumpTree(int nDepth = 0) const;
#endif
    virtual const SourceLine &GetSourceLine() const;
};

class CompoundToken : public ListToken<Token>
{
public:
    CompoundToken(int);
    CompoundToken(const CompoundToken &);
```

```
    virtual  CompoundToken();
    virtual CompoundToken *Copy() const;

    virtual MString GetText(SourceLine &sl) const;
#ifdef DEBUG
    virtual void DumpTree(int nDepth = 0) const;
#endif
};

class TextToken : public Token
{
    SourceLine line;
    MString strText;

public:
    TextToken(const SourceLine &,int, const char *);
    TextToken(const TextToken &);
    TextToken *Copy() const;

    virtual MString GetText(SourceLine &sl) const;
    const char *GetLiteralText() const;

    static TextToken *SwallowType(UserType *);

    virtual const SourceLine &GetSourceLine() const;
};

class CharToken : public Token
{
    SourceLine line;

public:
    CharToken(const SourceLine &, char);
    CharToken(const CharToken &);
    CharToken *Copy() const;

    virtual MString GetText(SourceLine &sl) const;
    virtual const SourceLine &GetSourceLine() const;
};

class DeclToken : public Token
{
private:
    SourceLine line;
    StorageClass *pStorage;
    Type *pType;
    MString strIdentifier;
    Token *pInitializer;
    Token *pBitfield;
    Token *pImplementation;
    Token *pAttribute;

protected:
    CompoundToken *FindParmList();

public:
    DeclToken(const SourceLine &, int, StorageClass *, Type *, const char *);
    DeclToken(const DeclToken &t);
    virtual  DeclToken();
    DeclToken *Copy() const;

    virtual MString GetText(SourceLine &sl) const;
#ifdef DEBUG
    virtual void DumpTree(int nDepth = 0) const;
#endif
    virtual DeclToken *GetDeclToken();
```

```
    void ComposeStorageClass(StorageClass *);
    void ComposeType(Type *);
    void ComposeSelf(DeclToken *);
    void ComposeQualifiers(Qualifiers *);
    void AdjoinBasicType(DeclToken *);
    static DeclToken *GenerateStructure(const SourceLine &, const char *,
                                        const char *, DeclListToken *);
    static DeclToken *GenerateStructure(const SourceLine &, const char *,
                                        const char *, DeclToken *);

    void SetInitializer(Token *);
    void SetImplementation(Token *);
    void SetBitfield(Token *);
    void SetAttribute(Token *);
    void ChangeIdentifier(const char *);
    const char *GetIdentifier() const;

    Type *ExtractType();

    static DeclToken *SwallowType(Type *p, const char * = "");
    static DeclToken *SwallowStorageClass(StorageClass *);
    static DeclToken *SwallowQualifiers(Qualifiers *);
    static DeclToken *SwallowIdentifier(const SourceLine &, const char *);
    static DeclToken *SwallowBitfield(Token *);
    static DeclToken *CreatePlaceholder(const SourceLine &, const char *);
    static DeclToken *BuildVarStruct(const SourceLine &, DeclToken *);
    static DeclToken *BuildInitialization(const SourceLine &,const char *,
                                          const char *, const char *);
#if 0
    static Token *BuildPayloadCalc(const char *);
#endif
    static DeclToken *BuildPackDecl(const SourceLine &sl, const char *,
                                    const char *, const char *);
    DeclListToken *BuildVarStructs();

#ifdef PREAMBLE
    static DeclListToken *GeneratePreambleDecls();
#endif
    DeclListToken *HandleDeclaration();
    DeclListToken *HandleDefinition();
//  DeclToken *GetDefinitionToken();
    DeclListToken *BuildRemoteFunctionDeclaration();
    DeclListToken *BuildRemoteFunctionDefinition();
    DeclToken *BuildRemotePackFunction(FunctionType *);
    DeclToken *BuildRemoteUnpackFunction(FunctionType *);

    bool IsSeparatorNeeded();
    virtual const SourceLine &GetSourceLine() const;
};

class DeclListToken : public ListToken<DeclToken>
{
public:
    enum { S_BETWEEN, S_PRE, S_POST, S_NONE, S_ASK };

private:
    char cSeparator;
    int nSeparatorPos;
    bool bEllipsis;

public:
    DeclListToken(int, char , int);
    DeclListToken(const DeclListToken &);
    virtual  DeclListToken();
    DeclListToken *Copy() const;
```

```
    void AddEllipsis();
    void SetSeparator(char, int);
    void NumberIdentifiers();
    void ConcatRight(DeclListToken *p);

    DeclListToken *HandleDeclarations();
    static DeclListToken *SwallowDeclToken(DeclToken *, char, int);
    static DeclListToken *DisperseDeclSpecs(DeclToken *, DeclListToken *,
                                            char, int);
    virtual MString GetText(SourceLine &sl) const;
#ifdef DEBUG
    virtual void DumpTree(int nDepth = 0) const;
#endif
};

class BoundedArrayToken : public Token
{
    SourceLine line;
    Token *pExpression;
    Token *pBound;

public:
    BoundedArrayToken(const SourceLine &,Token *, Token *);
    BoundedArrayToken(const BoundedArrayToken &);
    virtual  BoundedArrayToken();

    BoundedArrayToken *Copy() const;

    virtual MString GetText(SourceLine &sl) const;
    virtual const SourceLine &GetSourceLine() const;
};

class BoundedArrayListToken : public ListToken<BoundedArrayToken>
{
public:
    BoundedArrayListToken(int);
    BoundedArrayListToken(const BoundedArrayListToken &);
    virtual  BoundedArrayListToken();
    virtual BoundedArrayListToken *Copy() const;
    virtual MString GetText(SourceLine &sl) const;

    static BoundedArrayListToken *SwallowBoundedArray(BoundedArrayToken *);
#ifdef DEBUG
    virtual void DumpTree(int nDepth = 0) const;
#endif
};

class RemoteStartToken : public Token
{
    SourceLine line;
    MString strIdentifier;
    CompoundToken *pArgs;
    Token *pAssign;

public:
    RemoteStartToken(const SourceLine &sl, const char *, CompoundToken *,
                     Token *);
    RemoteStartToken(const RemoteStartToken &);
    virtual  RemoteStartToken();
    virtual RemoteStartToken *Copy() const;
    virtual MString GetText(SourceLine &sl) const;

    static RemoteStartToken *Create(TextToken *, CompoundToken *,
                                    Token * = NULL);
    virtual const SourceLine &GetSourceLine() const;
```

```
    const char *GetIdentifier();
    const CompoundToken *GetArgs();
    const Token *GetAssigned();
};

class RemoteStartListToken : public ListToken<RemoteStartToken>
{
public:
    RemoteStartListToken(int);
    RemoteStartListToken(const RemoteStartListToken &);
    RemoteStartListToken *Copy() const;
    virtual MString GetText(SourceLine &sl) const;

    static RemoteStartListToken *Swallow(RemoteStartToken *);
};

class DistForToken : public Token
{
    SourceLine line;
    Token *pExp1;
    Token *pExp2;
    Token *pExp3;
    RemoteStartToken *pBody;

public:
    DistForToken(const SourceLine &, Token *, Token *, Token *,
                 RemoteStartToken *);
    DistForToken(const DistForToken &);
    virtual  DistForToken();
    virtual DistForToken *Copy() const;
    virtual MString GetText(SourceLine &sl) const;
    virtual const SourceLine &GetSourceLine() const;
};

class DistBlockToken : public Token
{
    SourceLine line;
    RemoteStartListToken *pBody;

public:
    DistBlockToken(const SourceLine &, RemoteStartListToken *);
    DistBlockToken(const DistBlockToken &);
    virtual  DistBlockToken();
    virtual DistBlockToken *Copy() const;
    virtual MString GetText(SourceLine &sl) const;
    virtual const SourceLine &GetSourceLine() const;
};

#endif
```

B.7.17 dc2/type.h

```
#ifndef TYPE_H
#define TYPE_H 1

#include "qual.h"
#include "srcitem.h"

class Type;
class BasicType;
class CompoundType;
class UserType;
class DerivedType;
class FunctionType;
class PlaceholderType;
```

```
class Type : public SourceItem
{
private:
    SourceLine line;
    Qualifiers *pQual;

protected:
    MString GetQualifiers(SourceLine &sl) const;
    static MString MangleIdentifier(const char *);
    MString LineCorrect(SourceLine &) const;

public:
    Type(const SourceLine &);
    Type(const SourceLine &, Qualifiers *);
    Type(const Type &);
    virtual  Type();
    virtual void Render(SourceLine &, MString &) const = 0;
    virtual Type *Compose(Type *) = 0;
    virtual Type *Copy() const = 0;
    virtual void Mangle(MString &) const;

    void ComposeQualifiers(Qualifiers *);
    void ComposeQualifiers(Type *);

    virtual BasicType *GetBasicType();
    virtual DerivedType *GetDerivedType();
#ifdef DEBUG
    virtual void Dump() const = 0;
#endif
    virtual Type *ReplacePlaceholder(Type *);
    virtual FunctionType *GetFunctionType();
    virtual Type *GetDefinitionType();
    Type *HandleProcessType();
    virtual const SourceLine &GetSourceLine() const;
};

class BasicType : public Type
{
public:
    enum
    {
        T_NONE,
        T_VOID,
        T_CHAR,
        T_SHORT,
        T_INT,
        T_LONG,
        T_LLONG,
        T_FLOAT, T_DOUBLE, T_LDOUBLE,
        NUMBER_OF_TYPES
    };

    enum
    {
        T_NOSIGN, T_SIGNED, T_UNSIGNED,
        NUMBER_OF_SIGNS
    };
private:
    int nSimpleType;
    int nSign;
    static const char *szBasicTypes[];
    static const char *szBasicSigns[];
    static const char *szBasicMangles[];
    static const char *szSignMangles[];
```

```
protected:
    void InternalComposeSelf(BasicType *);

public:
    BasicType(const SourceLine &, Qualifiers *pq, int n, int s);
    BasicType(const SourceLine &, int n, int s);
    BasicType(const BasicType &);
    virtual void Render(SourceLine &sl, MString &) const;
    virtual void Mangle(MString &) const;
    virtual Type *Compose(Type *);

    virtual BasicType *GetBasicType();
#ifdef DEBUG
    virtual void Dump() const;
#endif
    virtual BasicType *Copy() const;

#ifdef PREAMBLE
    static void BuildSimpleStructs(DeclListToken *);
#endif
};

class CompoundType : public Type
{
public:
    enum
    {
        C_STRUCT = 1,
        C_UNION = 2,
        C_ENUM = 3
    };

private:
    int nClass;
    MString strName;
    DeclListToken *pContents;
    bool bNamed;

public:
    CompoundType(const SourceLine &, Qualifiers *pq, int n,
                 const char *s = NULL,
                 DeclListToken *p = NULL, bool b = false);
    CompoundType(const SourceLine &, int n, const char *s = NULL,
                 DeclListToken *p = NULL, bool b = false);
    CompoundType(const CompoundType &);
     CompoundType();
    virtual void Render(SourceLine &, MString &) const;
    virtual void Mangle(MString &) const;
    virtual Type *Compose(Type *);
    void SetName(const char *);
    void SetNameless();
    void SetContents(DeclListToken *);
#ifdef DEBUG
    virtual void Dump() const;
#endif
    virtual CompoundType *Copy() const;
    virtual Type *GetDefinitionType();
};

class UserType : public Type
{
    MString strName;

public:
    UserType(const SourceLine &, Qualifiers *pq, const char *s);
    UserType(const SourceLine &, const char *s);
```

```
    UserType(const UserType &);
    virtual void Render(SourceLine &, MString &) const;
    virtual void Mangle(MString &) const;
    virtual Type *Compose(Type *);
    const char *GetName() const;
#ifdef DEBUG
    virtual void Dump() const;
#endif
    virtual UserType *Copy() const;
};

class DerivedType : public Type
{
    Type *pApply;

protected:
    virtual void InternalComposeSelf(Type *);
    Type *GetNext();

public:
    DerivedType(const SourceLine &, Qualifiers *pq, Type *p);
    DerivedType(const SourceLine &, Type *p);
    DerivedType(const DerivedType &);
    virtual  DerivedType();
    virtual Type *Compose(Type *);
    virtual void Render(SourceLine &, MString &) const;
    virtual void Mangle(MString &) const;
#ifdef DEBUG
    virtual void Dump() const;
#endif
    virtual DerivedType *Copy() const = 0;
    virtual DerivedType *GetDerivedType();
    virtual Type *ReplacePlaceholder(Type *);
};

class PointerType : public DerivedType
{
public:
    PointerType(const SourceLine &, Qualifiers *pq, Type *p);
    PointerType(const SourceLine &, Type *p);
    PointerType(const PointerType &);
    virtual  PointerType();
    virtual void Render(SourceLine &, MString &) const;
    virtual void Mangle(MString &) const;
#ifdef DEBUG
    virtual void Dump() const;
#endif
    virtual PointerType *Copy() const;
};

class FunctionType : public DerivedType
{
    DeclListToken *pArgs;
public:
    FunctionType(const SourceLine &, Qualifiers *pq, Type *p, DeclListToken *r);
    FunctionType(const SourceLine &, Type *p, DeclListToken *r);
    FunctionType(const FunctionType &);
    virtual  FunctionType();
    virtual void Render(SourceLine &, MString &) const;
    virtual void Mangle(MString &) const;
#ifdef DEBUG
    virtual void Dump() const;
#endif
    virtual FunctionType *Copy() const;
    virtual FunctionType *GetFunctionType();
    Type *CopyReturnType();
```

```
    void MangleReturnType(MString &);
    DeclListToken *CopyParameterList() const;
    unsigned int GetParameterCount() const;
};

class ArrayType : public DerivedType
{
    Token *pBoundExpression;

public:
    ArrayType(const SourceLine &, Qualifiers *pq, Type *p, Token *r);
    ArrayType(const SourceLine &, Type *p, Token *r);
    ArrayType(const ArrayType &);
    virtual  ArrayType();
    virtual void Render(SourceLine &, MString &) const;
    virtual void Mangle(MString &) const;
#ifdef DEBUG
    virtual void Dump() const;
#endif
    virtual ArrayType *Copy() const;
};

class PlaceholderType : public Type
{
public:
    PlaceholderType(const SourceLine &);
    PlaceholderType(const PlaceholderType &);
    virtual void Render(SourceLine &, MString &) const;
    virtual void Mangle(MString &) const;
#ifdef DEBUG
    virtual void Dump() const;
#endif
    virtual Type *Compose(Type *);
    virtual PlaceholderType *Copy() const;
    virtual Type *ReplacePlaceholder(Type *);
};

class ProcessType : public Type
{
    Type *pInnerType;

public:
    ProcessType(const SourceLine &, Type *);
    ProcessType(const ProcessType &);
    virtual  ProcessType();
    virtual void Render(SourceLine &, MString &) const;
    virtual void Mangle(MString &) const;
#ifdef DEBUG
    virtual void Dump() const;
#endif
    virtual Type *Compose(Type *);
    virtual ProcessType *Copy() const;
};

#endif
```

B.7.18 dc2/storage.h

```
#ifndef STORAGE_H
#define STORAGE_H 1

#include "srcitem.h"

class StorageClass : public SourceItem
{
```

```
public:
    enum
    {
        S_NONE = 0,
        S_TYPEDEF = 1,
        S_EXTERN = 2,
        S_REMOTE = 4,
        S_STATIC = 8,
        S_INLINE = 16,
        S_AUTO = 32,
        S_REGISTER = 64,
        MAX_CLASS = 127
    };

private:
    SourceLine line;
    int nStorage;

    static const char *szClassNames[];

public:
    StorageClass(const SourceLine &, int = S_NONE);
    StorageClass(const StorageClass &);
    StorageClass *Copy();
    void Set(int);
    int Get() const;
    void ComposeSelf(StorageClass *);
    bool IsClass(int) const;
    void Render(SourceLine &, MString &) const;

    virtual const SourceLine &GetSourceLine() const;
};
#endif
```

B.7.19 dc2/qual.h

```
#ifndef QUAL_H
#define QUAL_H 1

#include "srcitem.h"

class Qualifiers : public SourceItem
{
public:
    enum { Q_NONE = 0, Q_CONST = 1, Q_VOLATILE = 2, Q_RESTRICT = 4,
           Q_ALL = Q_CONST | Q_VOLATILE };

private:
    SourceLine line;
    int nQualifiers;

public:
    Qualifiers(const SourceLine &, int = Q_NONE);
    Qualifiers(const Qualifiers &q);
    Qualifiers *Copy();

    MString GetText(SourceLine &) const;
    void Mangle(MString &) const;
//  void Set(int);
//  void Set(Qualifiers *);
    void Merge(int);
    void Merge(Qualifiers *);
    void Clear();
    virtual const SourceLine &GetSourceLine() const;
```

```
};
#endif
```

B.7.20 dc2/dc.y

```
%{
#include "project.h"
#include "dc2.h"
#include "srcitem.h"
#include "token.h"
#include "type.h"
#include "storage.h"
#include "tokid.h"

#define new DEBUG_NEW

bool bInTypedef = false;

int nInhibit = 0;

void HandleTypedef(const DeclToken *);

CompoundToken *NewNode(int,
                       Token *t1=NULL, Token *t2=NULL,
                       Token *t3=NULL, Token *t4=NULL,
                       Token *t5=NULL, Token *t6=NULL,
                       Token *t7=NULL, Token *t8=NULL);

#define yylex() Compiler::Instance()->Lex()
#define STR(A) #A
// #define Compose(A) Compose(A, __LINE__)

%}

%union {
    Token *token;
    TextToken *ttoken;
    CompoundToken *ctoken;
    DeclToken *dtoken;
    DeclListToken *ltoken;
    BoundedArrayToken *batoken;
    BoundedArrayListToken *baltoken;
    RemoteStartToken *rstoken;
    RemoteStartListToken *rsltoken;
    DistForToken *dftoken;
    DistBlockToken *dbtoken;

    Type *type;
    UserType *utype;
    CompoundType *ctype;

    StorageClass *sc;
    Qualifiers *qual;
}

%token NULLTOK
%token ';' '{' '}' ',' ':' '=' '(' ')' '[' ']' '.' '&' '!' ' ' '-' '+' '*'
%token '/' '%' '<' '>' '^' '|' '?'

%token CHAR SHORT INT LONG FLOAT DOUBLE VOID
%token SIGNED UNSIGNED CONST VOLATILE RESTRICT
%token STRUCT UNION ENUM
%token TYPE_NAME
```

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token ELLIPSIS
%token INLINE
%token PROCESS REMOTE DSTART DWAIT DSELECT DFOR DBLOCK

%token CASE DEFAULT IF SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%token ATTRIBUTE
%token ELSE

%type<token> NULLTOK
%type<token> ';' '{' '}' ',' ':' '=' '(' ')'
%type<token> '[' ']' '.' '&' '!' ' ' '-' '+' '*'
%type<token> '/' '%' '<' '>' '^' '|' '?'

%type<type> CHAR SHORT INT LONG FLOAT DOUBLE VOID
%type<type> SIGNED UNSIGNED
%type<ctype> STRUCT UNION ENUM
%type<utype> TYPE_NAME
%type<qual> CONST VOLATILE RESTRICT

%type<ttoken> IDENTIFIER
%type<token> CONSTANT STRING_LITERAL SIZEOF
%type<token> PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%type<token> AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%type<token> SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%type<token> XOR_ASSIGN OR_ASSIGN

%type<sc> TYPEDEF EXTERN STATIC AUTO REGISTER REMOTE INLINE

%type<token> ELLIPSIS
%type<token> PROCESS

%type<token> CASE DEFAULT IF SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%type<token> ATTRIBUTE
%type<ttoken> DSTART DWAIT DSELECT DFOR DBLOCK
%right<token> ELSE

%type<sc> storage_class_specifier

%type<ttoken> id_or_type
%type<token> primary_expression unary_expression unary_operator
%type<token> assignment_operator
%type<token> postfix_expression
%type<token> expression_statement statement labeled_statement
%type<token> compound_statement
%type<token> selection_statement iteration_statement jump_statement
%type<token> cast_expression multiplicative_expression
%type<token> additive_expression shift_expression relational_expression
%type<token> equality_expression logical_and_expression
%type<token> and_expression exclusive_or_expression inclusive_or_expression
%type<token> logical_or_expression conditional_expression assignment_expression
%type<token> expression constant_expression
%type<ltoken> declaration2
%type<ltoken> declaration
%type<token> initializer
%type<token> enter_scope leave_scope
%type<ltoken> translation_unit external_declaration external_declaration2
```

```
/* %type<token> opt_decl_list identifier_list */
%type<ltoken> function_definition

%type<ctoken> argument_expression_list
%type<ltoken> init_declarator_list_aftertype init_declarator_list_notype
%type<ctoken> initializer_list statement_list
%type<ltoken> struct_declaration_list
%type<ltoken> struct_declarator_list_aftertype struct_declarator_list_notype
%type<ltoken> parameter_list parameter_type_list
%type<ltoken> enumerator_list declaration_list
%type<ltoken> struct_declaration

%type<dtoken> declaration_specifiers_withtype declaration_specifiers_builtin
%type<dtoken> declaration_specifiers_typename declaration_specifiers_notype
%type<dtoken> declaration_specifiers
%type<dtoken> parameter_declaration parameter_declaration2
%type<dtoken> function_declaration
%type<dtoken> init_declarator init_declarator_notype
%type<dtoken> init_declarator2 init_declarator_notype2
%type<dtoken> enum_id_or_type

%type<ctype> struct_or_union

%type<dtoken> declarator_identifier
%type<dtoken> enumerator
%type<type> enum_specifier
%type<type> struct_or_union_specifier struct_or_union_specifier2
%type<type> type_specifier_builtin
%type<type> type_name
%type<type> process_specifier
%type<dtoken> sql_builtin sql_withtype sql_typename specifier_qualifier_list
%type<dtoken> pointer abstract_declarator direct_abstract_declarator
%type<dtoken> struct_declarator struct_declarator_notype
%type<dtoken> declarator declarator_notype
/*%type<dtoken> declarator2 declarator_notype2*/
%type<dtoken> direct_declarator_notype
%type<dtoken> array_bounds func_proto func_params empty_func_params

%type<token> attribute
%type<token> dc_expression dc_statement
%type<dftoken> dc_for
%type<dbtoken> dc_block
%type<rstoken> dc_invoke
%type<rsltoken> dc_invoke_list
%type<rstoken> dc_call
%type<baltoken> dc_selection_list
%type<batoken> dc_selection_arg

%type<qual> type_qualifier_list type_qualifier sql_notype

%start translation_unit
%%

/*
 * If we're dealing with a type pretending to be an identifier then
 * create a new TextToken containing its name and throw away the
 * old UserType
 */

id_or_type
        : IDENTIFIER
            { $$ = $1; }
        | TYPE_NAME
            { $$ = TextToken::SwallowType($1); }
        ;
```

```
enum_id_or_type
        : IDENTIFIER
            { $$ = DeclToken::SwallowIdentifier(Compiler::GetCurrentLine(),
                                                  $1->GetLiteralText());
              delete $1; }
        | TYPE_NAME
            { $$ = DeclToken::SwallowType($1); }
        ;

primary_expression
        : IDENTIFIER
            { $$ = $1; }
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
            { $$ = NewNode(EXPR, $1, $2, $3); }
        ;

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
            { $$ = NewNode(EXPR, $1, $2, $3, $4); }
        | postfix_expression '(' ')'
            { $$ = NewNode(EXPR, $1, $2, $3); }
        | postfix_expression '(' argument_expression_list ')'
            { $$ = NewNode(EXPR, $1, $2, $3, $4); }
        | postfix_expression '.' id_or_type
            { $$ = NewNode(EXPR, $1, $2, $3); }
        | postfix_expression PTR_OP id_or_type
            { $$ = NewNode(EXPR, $1, $2, $3); }
        | postfix_expression INC_OP
            { $$ = NewNode(EXPR, $1, $2); }
        | postfix_expression DEC_OP
            { $$ = NewNode(EXPR, $1, $2); }
        | dc_expression
        ;

argument_expression_list
        : assignment_expression
            { $$ = NewNode(EXPRLIST, $1); }
        | argument_expression_list ',' assignment_expression
            { $1->AdoptRight($2); $1->AdoptRight($3); $$ = $1; }
        ;

unary_expression
        : postfix_expression
        | INC_OP unary_expression
            { $$ = NewNode(EXPR, $1, $2); }
        | DEC_OP unary_expression
            { $$ = NewNode(EXPR, $1, $2); }
        | unary_operator cast_expression
            { $$ = NewNode(EXPR, $1, $2); }
        | SIZEOF unary_expression
            { $$ = NewNode(EXPR, $1, $2); }
        | SIZEOF '(' type_name ')'
            { $$ = NewNode(EXPR, $1, $2, DeclToken::SwallowType($3), $4); }
        ;

unary_operator
        : '&'
        | '*'
        | '+'
        | '-'
        | ' '
        | '!'
        ;
```

```
cast_expression
        : unary_expression
            { $$ = $1; }
        | '(' type_name ')' cast_expression
            { $$ = NewNode(EXPR, $1, DeclToken::SwallowType($2), $3, $4); }
        ;

multiplicative_expression
        : cast_expression
            { $$ = $1; }
        | multiplicative_expression '*' cast_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        | multiplicative_expression '/' cast_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        | multiplicative_expression '%' cast_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        ;

additive_expression
        : multiplicative_expression
            { $$ = $1; }
        | additive_expression '+' multiplicative_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        | additive_expression '-' multiplicative_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        ;

shift_expression
        : additive_expression
        | shift_expression LEFT_OP additive_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        | shift_expression RIGHT_OP additive_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        ;

relational_expression
        : shift_expression
        | relational_expression '<' shift_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        | relational_expression '>' shift_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        | relational_expression LE_OP shift_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        | relational_expression GE_OP shift_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        ;

equality_expression
        : relational_expression
        | equality_expression EQ_OP relational_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        | equality_expression NE_OP relational_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        ;

and_expression
        : equality_expression
        | and_expression '&' equality_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        ;

exclusive_or_expression
        : and_expression
        | exclusive_or_expression '^' and_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
```

```
        ;

inclusive_or_expression
        : exclusive_or_expression
        | inclusive_or_expression '|' exclusive_or_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        ;

logical_and_expression
        : inclusive_or_expression
        | logical_and_expression AND_OP inclusive_or_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        ;

logical_or_expression
        : logical_and_expression
        | logical_or_expression OR_OP logical_and_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        ;

conditional_expression
        : logical_or_expression
        | logical_or_expression '?' expression ':' conditional_expression
            { $$ = NewNode(EXPR, $1, $2, $3, $4, $5); }
        ;

assignment_expression
        : conditional_expression
        | unary_expression assignment_operator assignment_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        ;

assignment_operator
        : '='
        | MUL_ASSIGN
        | DIV_ASSIGN
        | MOD_ASSIGN
        | ADD_ASSIGN
        | SUB_ASSIGN
        | LEFT_ASSIGN
        | RIGHT_ASSIGN
        | AND_ASSIGN
        | XOR_ASSIGN
        | OR_ASSIGN
        ;

expression
        : assignment_expression
        | expression ',' assignment_expression
            { $$ = NewNode(EXPR, $1, $2, $3); }
        ;

constant_expression
        : conditional_expression
        ;

declaration : declaration2
{
    bInTypedef = false;
    $$ = $1->HandleDeclarations();
}

declaration2
        : declaration_specifiers ';'
            { $$ = DeclListToken::SwallowDeclToken($1, ';',
                                                   DeclListToken::S_POST);
```

```
            delete $2; }
        | declaration_specifiers_withtype init_declarator_list_aftertype ';'
            { $$ = DeclListToken::DisperseDeclSpecs($1, $2, ';',
                                                      DeclListToken::S_POST);
            delete $3; }
        | declaration_specifiers_notype init_declarator_list_notype ';'
            { $$ = DeclListToken::DisperseDeclSpecs($1, $2, ';',
                                                      DeclListToken::S_POST);
            delete $3; }
        ;

declaration_specifiers_withtype
        : declaration_specifiers_typename
        | declaration_specifiers_builtin
        ;

declaration_specifiers_builtin
        : storage_class_specifier declaration_specifiers_builtin
            { $2->ComposeStorageClass($1); $$ = $2; }
        | type_specifier_builtin
            { $$ = DeclToken::SwallowType($1); }
        | type_specifier_builtin declaration_specifiers_notype
            { $2->ComposeType($1); $$ = $2; }
        | type_specifier_builtin declaration_specifiers_builtin
            { $2->ComposeType($1); $$ = $2; }
        | type_qualifier declaration_specifiers_builtin
            { $2->ComposeQualifiers($1); $$ = $2; }
        ;

declaration_specifiers_typename
        : storage_class_specifier declaration_specifiers_typename
            { $2->ComposeStorageClass($1); $$ = $2; }
        | TYPE_NAME
            { $$ = DeclToken::SwallowType($1); }
        | TYPE_NAME declaration_specifiers_notype
            { $2->ComposeType($1); $$ = $2; }
        | type_qualifier declaration_specifiers_typename
            { $2->ComposeQualifiers($1); $$ = $2; }
        ;

declaration_specifiers_notype
        : storage_class_specifier
            { $$ = DeclToken::SwallowStorageClass($1); }
        | storage_class_specifier declaration_specifiers_notype
            { $2->ComposeStorageClass($1); $$ = $2; }
        | type_qualifier
            { $$ = DeclToken::SwallowQualifiers($1); }
        | type_qualifier declaration_specifiers_notype
            { $2->ComposeQualifiers($1); $$ = $2; }
        ;

declaration_specifiers
        : declaration_specifiers_withtype
        | declaration_specifiers_notype

init_declarator_list_aftertype
        : init_declarator
            { $$ = DeclListToken::SwallowDeclToken($1, ',',
                                                     DeclListToken::S_BETWEEN); }
        | init_declarator_list_aftertype ',' init_declarator
            { $1->AdoptRight($3); delete $2; $$ = $1; }
        ;

init_declarator_list_notype
        : init_declarator_notype
            { $$ = DeclListToken::SwallowDeclToken($1, ',',
```

```
                                                  DeclListToken::S_BETWEEN); }
        | init_declarator_list_notype ',' init_declarator
            { $1->AdoptRight($3); delete $2; $$ = $1; }
        ;

init_declarator : init_declarator2
{
    if (bInTypedef && !nInhibit)
    {
        /*TRACE2("(%d) Hit a typedefed declarator '%s'\n",
               Compiler::Instance()->GetCurrentLine(),
               (const char *)$1->GetIdentifier());*/
        HandleTypedef($1);
    }
    /*else
        TRACE2("(%d) Not a typedef declarator '%s'\n",
               Compiler::Instance()->GetCurrentLine(),
               (const char *)$1->GetIdentifier());*/
    $$ = $1;
}

init_declarator2
        : declarator
            { $$ = $1; }
        | declarator '=' initializer
            { $1->SetInitializer($3); delete $2; $$ = $1; }
        ;

init_declarator_notype : init_declarator_notype2
{
    if (bInTypedef && !nInhibit)
    {
        /*TRACE2("(%d) Hit a typedefed declarator '%s'\n",
               Compiler::Instance()->GetCurrentLine(),
               (const char *)$1->GetIdentifier()); */
        HandleTypedef($1);
    }
    /*else
        TRACE2("(%d) Not a typedef declarator '%s'\n",
               Compiler::Instance()->GetCurrentLine(),
               (const char *)$1->GetIdentifier());*/
    $$ = $1;
}

init_declarator_notype2
        : declarator_notype
            { $$ = $1; }
        | declarator_notype '=' initializer
            { $1->SetInitializer($3); delete $2; $$ = $1; }

storage_class_specifier
        : TYPEDEF { $$ = $1; bInTypedef = true; }
        | EXTERN
        | REMOTE
        | STATIC
        | INLINE
        | AUTO
        | REGISTER
        ;

type_specifier_builtin
        : VOID
        | CHAR
        | SHORT
        | INT
        | LONG
```

```
        | FLOAT
        | DOUBLE
        | SIGNED
        | UNSIGNED
        | struct_or_union_specifier
        | enum_specifier
        | process_specifier
        ;

struct_or_union_specifier
        : struct_or_union_specifier2 { $$ = $1; nInhibit--; }

struct_or_union_specifier2
        : struct_or_union id_or_type '{' struct_declaration_list '}'
            { $1->SetName($2->GetLiteralText()); $1->SetContents($4);
              $$ = $1; delete $2; delete $3; delete $5; }
        | struct_or_union '{' struct_declaration_list '}'
            { $1->SetNameless(); $1->SetContents($3); $$ = $1;
              delete $2; delete $4; }
        | struct_or_union id_or_type
            { $1->SetName($2->GetLiteralText()); $$ = $1; delete $2; }
        ;

struct_or_union
        : STRUCT { $$ = $1; nInhibit++; }
        | UNION { $$ = $1; nInhibit++; }
        ;

struct_declaration_list
        : struct_declaration
            { $$ = $1; }
        | struct_declaration_list struct_declaration
            { $1->ConcatRight($2); $$ = $1; }
        ;

struct_declaration
        : sql_withtype struct_declarator_list_aftertype ';'
            { $$ = DeclListToken::DisperseDeclSpecs($1, $2, ';',
                                                   DeclListToken::S_POST);
              delete $3; }
        | sql_notype struct_declarator_list_notype ';'
            { DeclToken *p = DeclToken::SwallowQualifiers($1);
              $$ = DeclListToken::DisperseDeclSpecs(p, $2, ';',
                                                   DeclListToken::S_POST);
              delete $3; }
        ;

/* sql = specifier_qualifier_list */

sql_withtype
        : sql_typename
        | sql_builtin
        ;

sql_builtin
        : type_specifier_builtin
            { $$ = DeclToken::SwallowType($1); }
        | type_specifier_builtin sql_notype
            { $1->ComposeQualifiers($2); $$ = DeclToken::SwallowType($1); }
        | type_specifier_builtin sql_builtin
            { $2->ComposeType($1); $$ = $2; }
        | type_qualifier sql_builtin
            { $2->ComposeQualifiers($1); $$ = $2; }
        ;

sql_typename
```

```
        : TYPE_NAME
            { $$ = DeclToken::SwallowType($1); }
        | TYPE_NAME sql_notype
            { $1->ComposeQualifiers($2); $$ = DeclToken::SwallowType($1); }
        | type_qualifier sql_typename
            { $2->ComposeQualifiers($1); $$ = $2; }
        ;

sql_notype
        : type_qualifier
        | type_qualifier sql_notype
            { $2->Merge($1); $$ = $2; }
        ;

/*
 * sql_notype is just a bunch of type qualifiers so we need to turn it into
 * a proper type here, implicit int rules again.
 */

specifier_qualifier_list
        : sql_withtype
        | sql_notype
            { $$ = DeclToken::SwallowQualifiers($1); }
        ;

struct_declarator_list_aftertype
        : struct_declarator
            { $$ = new DeclListToken(DECRLIST, ',', DeclListToken::S_BETWEEN);
              $$->AdoptRight($1); }
        | struct_declarator_list_aftertype ',' struct_declarator
            { $1->AdoptRight($3); $$ = $1; delete $2; }
        ;

struct_declarator_list_notype
        : struct_declarator_notype
            { $$ = new DeclListToken(DECRLIST, ',', DeclListToken::S_BETWEEN);
              $$->AdoptRight($1); }
        | struct_declarator_list_notype ',' struct_declarator
            { $1->AdoptRight($3); $$ = $1; delete $2; }
        ;

struct_declarator
        : declarator
        | ':' constant_expression
            { $$ = DeclToken::SwallowBitfield($2); delete $1; }
        | declarator ':' constant_expression
            { $1->SetBitfield($3); $$ = $1; delete $2; }
        ;

struct_declarator_notype
        : declarator_notype
        | ':' constant_expression
            { $$ = DeclToken::SwallowBitfield($2); delete $1; }
        | declarator_notype ':' constant_expression
            { $1->SetBitfield($3); $$ = $1; delete $2; }
        ;

/*
struct_declarator
        : declarator
        ;

struct_declarator_notype
        : declarator_notype
        ;
        */
```

```
enum_specifier
        : ENUM '{' enumerator_list '}'
            { $1->SetNameless(); $1->SetContents($3); $$ = $1;
              delete $2; delete $4; }
        | ENUM id_or_type '{' enumerator_list '}'
            { $1->SetName($2->GetLiteralText()); $1->SetContents($4); $$ = $1;
              delete $3; delete $5; }
        | ENUM id_or_type
            { $1->SetName($2->GetLiteralText()); $$ = $1; }
        ;

enumerator_list
        : enumerator
            { $$ = new DeclListToken(ENUMLIST, ',', DeclListToken::S_BETWEEN);
              $$->AdoptRight($1); }
        | enumerator_list ',' enumerator
            { $1->AdoptRight($3); delete $2; $$ = $1; }
        ;

enumerator
        : enum_id_or_type
        | enum_id_or_type '=' constant_expression
            { $1->SetInitializer($3); delete $2; $$ = $1; }
        ;

type_qualifier
        : CONST
        | VOLATILE
        | RESTRICT
        ;

declarator_identifier : IDENTIFIER
{
    /* if (bInTypedef && !nInhibit)
        bInTypedef = false, HandleTypedef($1); */
    $$ = DeclToken::CreatePlaceholder(Compiler::GetCurrentLine(),
                                      $1->GetLiteralText());
    delete $1;
}

/*declarator : declarator2
{
    if (bInTypedef && !nInhibit)
    {
        TRACE2("(%d) Hit a typedefed declarator '%s'\n",
               Compiler::Instance()->GetCurrentLine(),
               (const char *)$1->GetIdentifier());
        HandleTypedef($1);
    }
    else
        TRACE2("(%d) Not a typedef declarator '%s'\n",
               Compiler::Instance()->GetCurrentLine(),
               (const char *)$1->GetIdentifier());
    $$ = $1;
}*/

declarator
        : declarator_notype attribute
            { $1->SetAttribute($2); $$ = $1; }
        | declarator_notype
/*      | typed_declarator */
        ;

/* typed_declarator
        : pointer typed_direct_declarator
```

```
            { $$ = NewNode(DECR, $1, $2); }
        | typed_direct_declarator
        ;
        */

declarator_notype
        : pointer direct_declarator_notype
            { $1->ComposeSelf($2); $$ = $1; }
        | direct_declarator_notype
            { $$ = $1; }
        ;

/* This used to contain
        : '(' identifier_list ')'
            { $$ = NewNode(FUNCPARM, $1, $2, $3); }
   but my other code wouldn't work with K&R style function definitions
   any way.
*/

func_proto
        : func_params
        | empty_func_params
        ;

func_params
        : '(' parameter_type_list ')'
            { $$ = DeclToken::SwallowType(new FunctionType($1->GetSourceLine(),
                                                          NULL, $2));
              delete $1; delete $3; }

empty_func_params
        : '(' ')'
            { $$ = DeclToken::SwallowType(new FunctionType($1->GetSourceLine(),
                                                          NULL, NULL));
              delete $1; delete $2; }

array_bounds
        : '[' constant_expression ']'
            { $$ = DeclToken::SwallowType(new ArrayType($1->GetSourceLine(),
                                                       NULL, $2));
              delete $1; delete $3; }
        | '[' ']'
            { $$ = DeclToken::SwallowType(new ArrayType($1->GetSourceLine(),
                                                       NULL, NULL));
              delete $1; delete $2; }

/* Gonna be brave here and ignore the brackets, they are just to help the
   parser, I hope! */

direct_declarator_notype
        : declarator_identifier
            { $$ = $1; }
        | '(' declarator_notype ')'
            { $$ = $2; delete $1; delete $3; }
        | direct_declarator_notype array_bounds
            { $2->ComposeSelf($1); $$ = $2; }
        | direct_declarator_notype func_proto
            { $2->ComposeSelf($1); $$ = $2; }
        ;

/* typed_direct_declarator
        : '(' typed_declarator ')'
            { $$ = NewNode(****, $1, $2, $3); }
        | typed_direct_declarator '[' constant_expression ']'
            { $$ = NewNode(****, $1, $2, $3, $4); }
        | typed_direct_declarator '[' ']'
```

```
            { $$ = NewNode(****, $1, $2, $3); }
        | typed_direct_declarator '(' parameter_type_list ')'
            { $$ = NewNode(****, $1, $2, $3, $4); }
        | typed_direct_declarator '(' identifier_list ')'
            { $$ = NewNode(****, $1, $2, $3, $4); }
        | typed_direct_declarator '(' ')'
            { $$ = NewNode(****, $1, $2, $3); }
        | TYPE_NAME
        ;
        */

/* param_declarator
        : pointer param_direct_declarator
            { $$ = NewNode(****, $1, $2); }
        | param_direct_declarator
        ;
        */

/* param_direct_declarator
        : param_direct_declarator '[' constant_expression ']'
            { $$ = NewNode(****, $1, $2, $3, $4); }
        | param_direct_declarator '[' ']'
            { $$ = NewNode(****, $1, $2, $3); }
        | param_direct_declarator '(' parameter_type_list ')'
            { $$ = NewNode(****, $1, $2, $3, $4); }
        | param_direct_declarator '(' identifier_list ')'
            { $$ = NewNode(****, $1, $2, $3, $4); }
        | param_direct_declarator '(' ')'
            { $$ = NewNode(****, $1, $2, $3); }
        | TYPE_NAME
        ;
        */

pointer
        : '*'
            { $$ = DeclToken::SwallowType(new PointerType($1->GetSourceLine(),
                                                          NULL));
              delete $1; }
        | '*' type_qualifier_list
            { $$ = DeclToken::SwallowType(new PointerType($1->GetSourceLine(),
                                                          $2, NULL));
              delete $1; }
        | '*' pointer
            { $2->ComposeType(new PointerType($1->GetSourceLine(), NULL));
              delete $1; $$ = $2; }
        | '*' type_qualifier_list pointer
            { $3->ComposeType(new PointerType($1->GetSourceLine(), $2, NULL));
              delete $1; $$ = $3; }
        ;

type_qualifier_list
        : type_qualifier
            { $$ = $1; }
        | type_qualifier_list type_qualifier
            { $1->Merge($2); $$ = $1; }
        ;

parameter_type_list
        : parameter_list
            { $$ = $1; }
        | parameter_list ',' ELLIPSIS
            { $1->AddEllipsis(); $$ = $1; delete $2; delete $3; }
        ;

parameter_list
        : parameter_declaration
```

```
            { $$ = new DeclListToken(PARMLIST, ',', DeclListToken::S_BETWEEN);
              $$->AdoptRight($1); }
        | parameter_list ',' parameter_declaration
            { $1->AdoptRight($3); delete $2; $$ = $1; }

parameter_declaration : parameter_declaration2
{
    $$ = $1;
//  TRACE0("### Got a parameter declaration\n");
//  ASSERT($1);
//  $1->DumpTree();
//  TRACE0("### End of dump\n");
}

parameter_declaration2
        : declaration_specifiers_withtype declarator_notype
            { $1->ComposeSelf($2); $$ = $1; }
        | declaration_specifiers_withtype abstract_declarator
            { $1->ComposeSelf($2); $$ = $1; }
/*        | declaration_specifiers_withtype param_declarator */
        | declaration_specifiers_notype declarator_notype
            { $1->ComposeSelf($2); $$ = $1; }
        | declaration_specifiers_notype abstract_declarator
            { $1->ComposeSelf($2); $$ = $1; }
        | declaration_specifiers
            { $$ = $1; }
        ;

/* Only used for K&R declarations
identifier_list
        : IDENTIFIER
            { $$ = NewNode(IDLIST, $1); }
        | identifier_list ',' IDENTIFIER
            { $1->AdoptRight($2); $1->AdoptRight($3); $$ = $1; }
        ;
        */

type_name
        : specifier_qualifier_list
            { $$ = $1->ExtractType(); delete $1; }
        | specifier_qualifier_list abstract_declarator
            { $1->ComposeSelf($2); $$ = $1->ExtractType(); delete $1; }
        ;

abstract_declarator
        : pointer
            { $$ = $1; }
        | direct_abstract_declarator
        | pointer direct_abstract_declarator
            { $2->ComposeSelf($1); $$ = $2; }
        ;

direct_abstract_declarator
        : '(' abstract_declarator ')'
            { $$ = $2; delete $1; delete $3; }
        | array_bounds
            { $$ = $1; }
        | direct_abstract_declarator array_bounds
            { $1->ComposeSelf($2); $$ = $1; }
        | func_proto
            { $$ = $1; }
        | direct_abstract_declarator func_proto
            { $1->ComposeSelf($2); $$ = $1; }
        ;

initializer
```

```
        : assignment_expression
        | '{' initializer_list '}'
            { $$ = NewNode(INIT, $1, $2, $3); }
        | '{' initializer_list ',' '}'
            { $$ = NewNode(INIT, $1, $2, $3, $4); }
        ;

initializer_list
        : initializer
            { $$ = NewNode(INITLIST, $1); }
        | initializer_list ',' initializer
            { $1->AdoptRight($2); $1->AdoptRight($3); $$ = $1; }
        ;

statement
        : labeled_statement
        | compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        | dc_statement
        ;

labeled_statement
        : id_or_type ':' statement
            { $$ = NewNode(STMT, $1, $2, $3); }
        | CASE constant_expression ':' statement
            { $$ = NewNode(STMT, $1, $2, $3, $4); }
        | DEFAULT ':' statement
            { $$ = NewNode(STMT, $1, $2, $3); }
        ;

enter_scope : { Compiler::Instance()->EnterScope(); }
leave_scope : { Compiler::Instance()->LeaveScope(); }

compound_statement
        : '{' '}'
            { $$ = NewNode(STMT, $1, $2); }
        | '{' enter_scope statement_list leave_scope '}'
            { $$ = NewNode(STMT, $1, $3, $5); }
        | '{' enter_scope declaration_list leave_scope '}'
            { $$ = NewNode(STMT, $1, $3, $5); }
        | '{' enter_scope declaration_list statement_list leave_scope '}'
            { $$ = NewNode(STMT, $1, $3, $4, $6); }
        ;

declaration_list
        : declaration
            { $$ = $1; }
        | declaration_list declaration
            { $1->ConcatRight($2); $$ = $1;}
        ;

statement_list
        : statement
            { $$ = NewNode(STMTLIST, $1); }
        | statement_list statement
            { $1->AdoptRight($2); $$ = $1; }
        ;

expression_statement
        : ';'
        | expression ';'
            { $$ = NewNode(STMT, $1, $2); }
        ;
```

```
selection_statement
        : IF '(' expression ')' statement %prec ELSE
            { $$ = NewNode(STMT, $1, $2, $3, $4, $5); }
        | IF '(' expression ')' statement ELSE statement
            { $$ = NewNode(STMT, $1, $2, $3, $4, $5, $6, $7); }
        | SWITCH '(' expression ')' statement
            { $$ = NewNode(STMT, $1, $2, $3, $4, $5); }
        ;

iteration_statement
        : WHILE '(' expression ')' statement
            { $$ = NewNode(STMT, $1, $2, $3, $4, $5); }
        | DO statement WHILE '(' expression ')' ';'
            { $$ = NewNode(STMT, $1, $2, $3, $4, $5, $6, $7); }
        | FOR '(' expression_statement expression_statement ')' statement
            { $$ = NewNode(STMT, $1, $2, $3, $4, $5, $6); }
        | FOR '(' expression_statement expression_statement expression ')'
          statement
            { $$ = NewNode(STMT, $1, $2, $3, $4, $5, $6, $7); }
        ;

jump_statement
        : GOTO id_or_type ';'
            { $$ = NewNode(STMT, $1, $2, $3); }
        | CONTINUE ';'
            { $$ = NewNode(STMT, $1, $2); }
        | BREAK ';'
            { $$ = NewNode(STMT, $1, $2); }
        | RETURN ';'
            { $$ = NewNode(STMT, $1, $2); }
        | RETURN expression ';'
            { $$ = NewNode(STMT, $1, $2, $3); }
        ;

/* There's no tree left here, so we don't do anything with it */
translation_unit
        : external_declaration
        | translation_unit external_declaration
        ;

external_declaration : external_declaration2
{
    Compiler::Instance()->Output($1);

    ASSERT($1);
//  TRACE0("$$$ Dumping external_declaration\n");
//  $1->DumpTree();
//  TRACE0("$$$ End dump\n");

//  TRACE1("Waving goodbye to tokens at [%p]\n", $1);
    delete $1;
//  TRACE0("Done waving\n");
    $$ = NULL;
}

/* Declaration is a dtoken but is converted to a token */

external_declaration2
        : function_definition
        | declaration
            { $$ = $1; }
        ;

/* Only used for K&R C
opt_decl_list
```

```
        :
            { $$ = new TextToken(NULLTOK, ""); }
        | declaration_list
        ;
        */

function_declaration
    : declaration_specifiers_notype declarator_notype
        { $1->ComposeSelf($2); $$ = $1; }
    | declaration_specifiers_withtype declarator
        { $1->ComposeSelf($2); $$ = $1; }
    | declarator
        { $$ = $1; }
    ;

/* This used to have opt_decl_list before the compound_statement but since
 * we're not supporting K&R style function definitions it went away.
 */
function_definition
        : function_declaration compound_statement
            { $1->SetImplementation($2); $$ = $1->HandleDefinition(); }
        ;

process_specifier
        : PROCESS '<' type_name '>'
            { $$ = new ProcessType($1->GetSourceLine(), $3);
              delete $1; delete $2; delete $4; }
        ;

attribute
        : ATTRIBUTE
        ;

dc_call
        : IDENTIFIER '(' argument_expression_list ')'
            { $$ = RemoteStartToken::Create($1, $3); delete $2; delete $4; }
        ;

dc_invoke
        : unary_expression '=' IDENTIFIER '(' argument_expression_list ')' ';'
            { $$ = RemoteStartToken::Create($3, $5, $1);
              delete $2; delete $4; delete $6; delete $7; }
        ;

dc_invoke_list
        : dc_invoke
            { $$ = RemoteStartListToken::Swallow($1); }
        | dc_invoke_list dc_invoke
            { $1->AdoptRight($2); $$ = $1; }
        ;


dc_expression
        : DSTART dc_call
            { $$ = $2; delete $1; }
        | DWAIT '(' expression ')'
            { $$ = Compiler::GenerateRemoteWait($1->GetSourceLine(), $3);
              delete $1; delete $2; delete $4; }
        | DSELECT '(' dc_selection_list ')'
           { $$ = Compiler::GenerateRemoteSelect($1->GetSourceLine(), $3);
             delete $1; delete $2; delete $4; }
     /* | DKILL '(' dc_selection_list ')' not supported */
        ;

dc_selection_list
        : dc_selection_arg
```

```
            { $$ = BoundedArrayListToken::SwallowBoundedArray($1); }
        | dc_selection_list ',' dc_selection_arg
            { $1->AdoptRight($3); $$ = $1; delete $2; }
        ;

dc_selection_arg
        : assignment_expression ':' constant_expression
            { $$ = new BoundedArrayToken($2->GetSourceLine(), $1, $3);
              delete $2; }
        | assignment_expression
            { $$ = new BoundedArrayToken($1->GetSourceLine(), $1, NULL); }
        ;

dc_for
        : DFOR '(' expression_statement expression_statement expression ')'
          dc_invoke
            { $$ = new DistForToken($1->GetSourceLine(), $3, $4, $5, $7);
              delete $1; delete $2; delete $6; }
        ;

dc_block
        : DBLOCK '{' dc_invoke_list '}'
            { $$ = new DistBlockToken($1->GetSourceLine(), $3);
              delete $1; delete $2; delete $4; }
        ;

dc_statement
        : dc_block
            { $$ = $1; }
        | dc_for
            { $$ = $1; }
        ;

%%
#include <stdio.h>

void HandleTypedef(const DeclToken *tok)
{
    Compiler::Instance()->AddSymbol(NS_GLOBAL, tok->GetIdentifier());
    ASSERT(Compiler::Instance()->IsSymbol(NS_GLOBAL, tok->GetIdentifier()));
}

#define ADOPT(A) if (A) t->AdoptRight(A);

CompoundToken *NewNode(int nType,
               Token *t1, Token *t2,
               Token *t3, Token *t4,
               Token *t5, Token *t6,
               Token *t7, Token *t8)
{
    CompoundToken *t = new CompoundToken(nType);
    ADOPT(t1);
    ADOPT(t2);
    ADOPT(t3);
    ADOPT(t4);
    ADOPT(t5);
    ADOPT(t6);
    ADOPT(t7);
    ADOPT(t8);
    return t;
}
```

B.7.21 dc2/dc.l

```
D                       [0-9]
L                       [a-zA-Z_]
H                       [a-fA-F0-9]
E                       [Ee][+-]?{D}+
FS                      (f|F|l|L)
IS                      (u|U|l|L)*
WH                      [ \t\v\f\n]+
%x COMMENT
%{
#include "project.h"
#include "dc2.h"
#include "dc.tab.h"

#define new DEBUG_NEW

void HandleIdentifierOrType();
inline void Count()
{
    Compiler::Instance()->Count();
}

inline bool IsSymbol(Namespace ns, const char *s)
{
    return Compiler::Instance()->IsSymbol(ns, s);
}
#define gcl Compiler::GetCurrentLine()

#define return_text(X) do { yylval.token = new TextToken(gcl, (X), yytext); \
                           return(X); } while(0)
#define return_char(X) do { yylval.token = new CharToken(gcl, (X)); \
                           return(X); } while(0)

#define return_basictype(X) do { \
    yylval.type = new BasicType(gcl, BasicType::T_##X, BasicType::T_NOSIGN); \
    return (X); } while(0)

#define return_compoundtype(X) do { \
    yylval.ctype = new CompoundType(gcl, CompoundType::C_##X); \
    return (X); } while (0)

#define return_utype(X) do { \
    yylval.utype = new UserType(gcl, yytext); return (X); } while (0)

#define return_qual(X) do { \
    yylval.qual = new Qualifiers(gcl, Qualifiers::Q_##X); \
    return (X); } while (0)

#define return_storage(X) do { \
    yylval.sc = new StorageClass(gcl, StorageClass::S_##X); \
    return (X); } while (0)

#define return_sign(X) do { \
    yylval.type = new BasicType(gcl, BasicType::T_NONE, BasicType::T_##X); \
    return (X); } while (0)
%}

%%
"/*"                    { BEGIN(COMMENT); }
<COMMENT>.              {}
<COMMENT>"*/"           { BEGIN(0); }

"char"                  { Count(); return_basictype(CHAR); }
"double"                { Count(); return_basictype(DOUBLE); }
"float"                 { Count(); return_basictype(FLOAT); }
"int"                   { Count(); return_basictype(INT); }
```

```
"long"                  { Count(); return_basictype(LONG); }
"void"                  { Count(); return_basictype(VOID); }
"short"                 { Count(); return_basictype(SHORT); }

"signed"                { Count(); return_sign(SIGNED); }
"__signed__"            { Count(); return_sign(SIGNED); }
"unsigned"              { Count(); return_sign(UNSIGNED); }

"struct"                { Count(); return_compoundtype(STRUCT); }
"union"                 { Count(); return_compoundtype(UNION); }
"enum"                  { Count(); return_compoundtype(ENUM); }

"auto"                  { Count(); return_storage(AUTO); }
"extern"                { Count(); return_storage(EXTERN); }
"inline"                { Count(); return_storage(INLINE); }
"register"              { Count(); return_storage(REGISTER); }
"remote"                { Count(); return_storage(REMOTE); }
"static"                { Count(); return_storage(STATIC); }
"typedef"               { Count(); return_storage(TYPEDEF); }
"__inline"              { Count(); return_storage(INLINE); }
"__inline__"            { Count(); return_storage(INLINE); }

"const"                 { Count(); return_qual(CONST); }
"__const"               { Count(); return_qual(CONST); }
"volatile"              { Count(); return_qual(VOLATILE); }
"__volatile"            { Count(); return_qual(VOLATILE); }
"__volatile__"          { Count(); return_qual(VOLATILE); }

"break"                 { Count(); return_text(BREAK); }
"case"                  { Count(); return_text(CASE); }
"continue"              { Count(); return_text(CONTINUE); }
"default"               { Count(); return_text(DEFAULT); }
"do"                    { Count(); return_text(DO); }
"dblock"                { Count(); return_text(DBLOCK); }
"dfor"                  { Count(); return_text(DFOR); }
"dselect"               { Count(); return_text(DSELECT); }
"dstart"                { Count(); return_text(DSTART); }
"dwait"                 { Count(); return_text(DWAIT); }
"else"                  { Count(); return_text(ELSE); }
"for"                   { Count(); return_text(FOR); }
"goto"                  { Count(); return_text(GOTO); }
"if"                    { Count(); return_text(IF); }
"process"               { Count(); return_text(PROCESS); }
"return"                { Count(); return_text(RETURN); }
"sizeof"                { Count(); return_text(SIZEOF); }
"switch"                { Count(); return_text(SWITCH); }
"while"                 { Count(); return_text(WHILE); }

"__attribute__"({WH})?"(".*")"  { Count(); return_text(ATTRIBUTE); }

{L}({L}|{D})*           { Count();
                          if (IsSymbol(NS_GLOBAL, YYText()))
                              return_utype(TYPE_NAME);
                          else
                              return_text(IDENTIFIER); }

0[xX]{H}+{IS}?          { Count(); return_text(CONSTANT); }
0{D}+{IS}?              { Count(); return_text(CONSTANT); }
{D}+{IS}?               { Count(); return_text(CONSTANT); }
'(\\.|[^\\'])+'         { Count(); return_text(CONSTANT); }

{D}+{E}{FS}?            { Count(); return_text(CONSTANT); }
{D}*"."{D}+({E})?{FS}?  { Count(); return_text(CONSTANT); }
{D}+"."{D}*({E})?{FS}?  { Count(); return_text(CONSTANT); }

\"(\\.|[^\\"])*\"       { Count(); return_text(STRING_LITERAL); }
```

```
"..."                   { Count(); return_text(ELLIPSIS); }
">>="                   { Count(); return_text(RIGHT_ASSIGN); }
"<<="                   { Count(); return_text(LEFT_ASSIGN); }
"+="                    { Count(); return_text(ADD_ASSIGN); }
"-="                    { Count(); return_text(SUB_ASSIGN); }
"*="                    { Count(); return_text(MUL_ASSIGN); }
"/="                    { Count(); return_text(DIV_ASSIGN); }
"%="                    { Count(); return_text(MOD_ASSIGN); }
"&="                    { Count(); return_text(AND_ASSIGN); }
"^="                    { Count(); return_text(XOR_ASSIGN); }
"|="                    { Count(); return_text(OR_ASSIGN); }
">>"                    { Count(); return_text(RIGHT_OP); }
"<<"                    { Count(); return_text(LEFT_OP); }
"++"                    { Count(); return_text(INC_OP); }
"--"                    { Count(); return_text(DEC_OP); }
"->"                    { Count(); return_text(PTR_OP); }
"&&"                    { Count(); return_text(AND_OP); }
"||"                    { Count(); return_text(OR_OP); }
"<="                    { Count(); return_text(LE_OP); }
">="                    { Count(); return_text(GE_OP); }
"=="                    { Count(); return_text(EQ_OP); }
"!="                    { Count(); return_text(NE_OP); }
";"                     { Count(); return_char(';'); }
"{"                     { Count(); return_char('{'); }
"}"                     { Count(); return_char('}'); }
","                     { Count(); return_char(','); }
":"                     { Count(); return_char(':'); }
"="                     { Count(); return_char('='); }
"("                     { Count(); return_char('('); }
")"                     { Count(); return_char(')'); }
"["                     { Count(); return_char('['); }
"]"                     { Count(); return_char(']'); }
"."                     { Count(); return_char('.'); }
"&"                     { Count(); return_char('&'); }
"!"                     { Count(); return_char('!'); }
" "                     { Count(); return_char(' '); }
"-"                     { Count(); return_char('-'); }
"+"                     { Count(); return_char('+'); }
"*"                     { Count(); return_char('*'); }
"/"                     { Count(); return_char('/'); }
"%"                     { Count(); return_char('%'); }
"<"                     { Count(); return_char('<'); }
">"                     { Count(); return_char('>'); }
"^"                     { Count(); return_char('^'); }
"|"                     { Count(); return_char('|'); }
"?"                     { Count(); return_char('?'); }

^{WH}?#.*\n         { Count(); Compiler::HandleDirective(yytext); }
{WH}            { Count(); }

.                       { yyerror("Invalid character in input"); }

%%

int yywrap()
{
    return 1;
}

#if 0
static void Comment()
{
    int c, c1;

loop:
```

```
    while ((c = yyinput()) != '*' && c != EOF)
        if (c == '\n')
            HandleGubbins("\n");

    if ((c1 = yyinput()) != '/' && c != EOF)
    {
        unput(c1);
        goto loop;
    }
}
#endif
```

B.7.22 dc2/dcc

```
#!/usr/local/bin/perl

$dc = "dc2";
$cc = "gcc";

$link = 1;
$compile = 1;
$translate = 1;
$preprocess = 1;

$verbose = 0;

$ccopts = " ";
$cppopts = " ";
$ldopts = " ";
$aout = "";
$path = "";

$tempfile = "/tmp/dcc$$";

$linklist = "";

$local_cppopts = "-D_REENTRANT -D__dc__ ";
$local_ccopts = "-fpic ";
$local_ldopts = "-shared ";

sub Report
{
    printf("Error: $_[0]\n");
}

sub Run
{
    ($verbose == 1) && (print "$_[0]\n");
    system $_[0];
    if ($? == 0)
    {
    return 1;
    }
    else
    {
    print "Failed to execute\n";
    return 0;
    }
}

sub ProcessFile
{
    local ($basename, $input, $output);
    local ($local_link, $local_compile, $local_translate, $local_preprocess)
    = ($link, $compile, $translate, $preprocess);
```

```
    $basename = $_[0];
    $input = $_[0];
    $output = "wrong";

  SWITCH:
    {
    ($basename =  /^(.*)\.dc/) && ($basename = $1, last SWITCH);
    ($basename =  /^(.*)\.di/) &&
        ($basename = $1, $local_preprocess = 0, last SWITCH);
    ($basename =  /^(.*)\.c/) &&
        ($basename = $1, $local_preprocess = $local_translate = 0,
         last SWITCH);
    ($basename =  /^(.*)\.o/) &&
        ($basename = $1, $local_preprocess = $local_translate =
         $local_compile = 0, last SWITCH);
    }
    ($aout eq "") && ($aout = "$basename.so");

    # Preprocess the input

    if ($local_preprocess)
    {
    $cmd .= "$cc -E -xc $cppopts$local_cppopts$input ";

    ($local_translate) && ($cmd .= ">$tempfile.di");

    (&Run($cmd)) || (return 0);
    }

    # Now translate it
    if ($local_translate)
    {
    $cmd = "$path$dc $dcopts";
    if ($local_preprocess)
    {
        $cmd .= "$tempfile.di ";
    }
    else
    {
        $cmd .= "$input ";
    }

    if ($local_compile)
    {
        $file = "$tempfile.gen.c"
    }
    else
    {
        $file = "$basename.gen.c"
    }
    $cmd .= ">$file";

    (&Run($cmd)) || (return 0);
#   (&Run("indent -kr $file")) || (return 0);
    }

    # Now compile it
    if ($local_compile)
    {
    $cmd = "$cc $cppopts$local_cppopts$ccopts$local_ccopts-x c -c ";
    $cmd .= "-o $basename.o ";
    if ($local_translate)
    {
        $cmd .= "$tempfile.gen.c ";
    }
```

```
    else
    {
        $cmd .= $input;
    }
    (&Run($cmd)) || (return 0);
    }
    if ($local_link)
    {
    $linklist .= " $basename.o";
    }
    return 1;
}

sub Link
{
    if ($link)
    {
    $cmd = "$cc $ccopts$local_ccopts$ldopts$local_ldopts-o $aout ";
    $cmd .= "$linklist";
    &Run($cmd);
    }
}

sub HandleDefine
{
    $cppopts .= "$_[0]$_[1] ";
}

$define = 0;

foreach $i (@ARGV)
{
    if ($define)
    {
    &HandleDefine($define, $i);
    $define = 0;
    }
    else
    {
    if ($i =  /^-(.*)/)
    {
        $opt = $1;
      SWITCH:
        {
        ($opt eq "c") && ($link = 0, last SWITCH);
        ($opt eq "d") && ($compile = $link = 0, last SWITCH);
        ($opt eq "E") && ($translate = $compile = $link = 0, last SWITCH);
        ($opt eq "ansi") && ($cppopts .= "-ansi ", last SWITCH);
        ($opt eq "pedantic") && ($cppopts .= "-pedantic ", last SWITCH);
        ($opt =  /^W(.*)/) && ($cppopts .= "-W$1 ", last SWITCH);
        ($opt =  /^o(.*)/) && ($aout = $1, last SWITCH);
        ($opt =  /^B(.*)/) && ($path = $1, last SWITCH);
        ($opt eq "D") && ($define = "-D", last SWITCH);
        ($opt eq "U") && ($define = "-U", last SWITCH);
        ($opt =  /^D(.*)/) && (&HandleDefine("-D", $1), last SWITCH);
        ($opt =  /^U(.*)/) && (&HandleDefine("-U", $1), last SWITCH);
        ($opt =  /^l(.*)/) && ($ldopts .= "-l$1 ", last SWITCH);
        ($opt =  /^L(.*)/) && ($ldopts .= "-L$1 ", last SWITCH);
        ($opt =  /^g(.*)/) && ($ccopts .= "-g$1 ", last SWITCH);
        ($opt =  /^O(.*)/) && ($ccopts .= "-O$1 ", last SWITCH);
        ($opt =  /^v/) && ($verbose = 1, last SWITCH);
        printf(STDERR "Unrecognised option -$opt\n");
        }
    }
    else
    {
```

```
        ($path !  /^$/) && ($path !  "/\$") && ($path .= '/');
        (&ProcessFile($i)) || (exit(1));
    }
    }
}

&Link();

unlink <$tempfile*>;
```

B.7.23 dc2/dc2seqc

```
#!/usr/local/bin/perl

$dselect1 = "({static int _dct_c;";
while(<>)
{
    chop;
  SWITCH:
    {
    ($_ =  /^(.*)process[ \t]*<([^>]*)>(.*)$/)
        && (printf("$1 $2 $3\n"), last SWITCH);
    ($_ =  /^(.*)dstart(.*)$/)
        && (printf("$1 $2\n"), last SWITCH);
    ($_ =  /^(.*)remote(.*)$/)
        && (printf("$1 $2\n"), last SWITCH);
    ($_ =  /^(.*)dwait(.*)$/)
        && (printf("$1 $2\n"), last SWITCH);
    ($_ =  /^(.*)dselect[ \t]*\([^:]+:(.*)[ \t]*\)(.*)$/)
        && (printf("$1 ({static int _dct_c; (_dct_c >= ($2))"),
        printf("? (_dct_c = 0) : _dct_c++; }) $3\n"),
        last SWITCH);
    print "$_\n";
    }
}
```

B.8 Runtime System

B.8.1 dcd2/Makefile.in

```
# Makefile.in for project/dcd

srcdir = @srcdir@
VPATH = @srcdir@
TOP = ..
THIS=dcd2
include $(TOP)/Make.rules

SUBDIRS=test

SRC = main.c exports.c hosts.c dist.c bucket.c
HEADERS = dist.h hosts.h
OBJ1= $(SRC:.c=.o)
OBJ = $(OBJ1:.C=.o)
TARGET = dcd

CFLAGS += $(CFLAGS_SHARED) $(CFLAGS_THREAD)
LDFLAGS += $(LDFLAGS_THREAD) $(LD_DYNAMIC)

LIBS += $(LIBDL) $(LIBPTHREAD) $(LIBSOCKET)

all : bin TAGS
```

```
dep :

bin : bin_this
    for i in $(SUBDIRS); do $(MAKE) -C $$i bin; done

doc : doc_this
    for i in $(SUBDIRS); do $(MAKE) -C $$i doc; done

clean : clean_this
    for i in $(SUBDIRS); do $(MAKE) -C $$i clean; done

distclean : distclean_this
    for i in $(SUBDIRS); do $(MAKE) -C $$i distclean; done

all_this : bin_this

bin_this : $(TARGET)

clean_this :
    $(RM) $(TARGET) $(OBJ) *  .depend pretty.tex

distclean_this : clean_this
    $(RM) Makefile

doc_this : pretty.tex

install : install.dirs install.bin

install.dirs :
    $(INSTALL) -d $(bindir)

install.bin : $(TARGET)
    $(INSTALL) $(TARGET) $(bindir)

$(TARGET) : $(OBJ) $(PROJLIB)
    $(LDXX) $(LDFLAGS) -o $(TARGET) $(OBJ) $(LIBS)

pretty.tex : Makefile.in $(SRC) $(HEADERS)
    $(TEXIFY) $^ > pretty.tex

TAGS : $(SRC) $(INCLUDE)
    $(ETAGS) $^
    $(80COL) $^

.depend : $(SRC)
    $(DEP) $(SRC) > .depend

Makefile : Makefile.in $(TOP)/config.status
    cd $(TOP) && CONFIG_FILES=$(THIS)/Makefile CONFIG_HEADERS= ./config.status

-include .depend
```

B.8.2 dcd2/main.c

```
#include "project.h"
#include "osthreads.h"
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <dlfcn.h>
#include <stdio.h>
#include <limits.h>
```

```
#include <sys/param.h>
#include <unistd.h>
#include <string.h>
#include "remote.h"
#include "dist.h"
#include "hosts.h"
#include <sys/utsname.h>

typedef int (*pfn)(int, char **, char **);

char _dc_ObjectName[PATH_MAX];

pid_t _dc_pid;

#define NONE 0
#define MASTER 1
#define SLAVE 2
#define RSH 4
#define PING 8
#define PINGHOSTS 9
#define KILL 16
#define KILLHOSTS 17

int _dci_doping(int, char **);
int _dci_dokill(int, char **);
int _dci_pinghosts();
int _dci_killhosts();
void set_limits(void);

static int call_main(int, char *[], char *[]);

int main(int argc, char *argv[], char *envp[])
{
#if !CONCURRENT
    int j;
    int mode = NONE;
#endif
    _dc_pid = getpid();

    if (argc < 2)
    {
        fprintf(stderr, "Usage dcd [options] objectfile\n");
        return 1;
    }

    set_limits();

    /* Skip past program name */
    argv++, argc--;

#if !CONCURRENT
    while(argc)
    {
        if ((*argv)[0] == '-')
        {
            for(j = 1; j < strlen(*argv); j++)
            {
                switch((*argv)[j])
                {
                case 's':
                    /* Start up in slave mode */
                    mode = SLAVE;
                    break;
                case 'm':
                    mode = MASTER;
                    break;
```

```
                case 'r':
                    mode = RSH;
                    break;
                case 'p':
                    mode = PINGHOSTS;
                    break;
                case 'P':
                    mode = PING;
                    break;
                case 'k':
                    mode = KILLHOSTS;
                    break;
                case 'K':
                    mode = KILL;
                    break;
                default:
                    fprintf(stderr, "Unrecognised option: %s\n", *argv);
                    break;
                }
            }
        }
        else
            break;
        argv++, argc--;
    }

    if (mode == PING)
        return _dci_doping(argc, argv);
    if (mode == KILL)
        return _dci_dokill(argc, argv);

#endif /* !CONCURRENT */
    if (!argc)
    {
        fprintf(stderr, "Must supply object file\n");
        exit(4);
    }

    /* Must be an object name */
    /* If we don't start with a / then apply the current directory */
    if ((*argv)[0] != '/')
    {
        getcwd(_dc_ObjectName, sizeof(_dc_ObjectName));
        strcat(_dc_ObjectName, "/");
        strcat(_dc_ObjectName, *argv);
    }
    else
        strcpy(_dc_ObjectName, *argv);
    /*TRACE1("Object file is %s\n", _dc_ObjectName);*/

#if !CONCURRENT
    if (_dcn_findhosts())
        exit(6);

    if (mode == PINGHOSTS)
        return _dci_pinghosts();
    else if (mode == KILLHOSTS)
        return _dci_killhosts();
    else if (mode == RSH)
    {
        if (_dcn_rshslaves())
            exit(7);
        return 0;
    }

    if (mode == MASTER)
```

```
    {
        return call_main(argc, argv, envp);
    }
    else if (mode == SLAVE)
    {
#ifdef CAN_DAEMON
        pid_t pid;
        int fd;
        struct utsname un;
        char szFile[PATH_MAX];

        strcpy(szFile, _dc_ObjectName);
        if (!strcmp(&szFile[strlen(szFile) - 3], ".so"))
            szFile[strlen(szFile) - 2] = '\0';
        uname(&un);
        strcat(szFile, un.nodename);
        strcat(szFile, ".log");
        fd = open(szFile, O_RDWR | O_APPEND | O_CREAT, 0666);
        if (fd == -1)
        {
            fprintf(stderr, "Unable to open logfile \'%s\'\n", szFile);
            exit(7);
        }

        fflush(stdout);
        fflush(stderr);
        pid = fork();
        if (pid == -1)
        {
            fprintf(stderr, "Unable to fork\n");
            exit(7);
        }
        if (pid == 0)
        {
            /* Child */
            _dc_pid = getpid();
            close(0);
            close(1);
            close(2);
            setsid();
            dup(fd);
            dup(fd);
            dup(fd);
            close(fd);
            fprintf(stderr, "\n** New log file opened\n");
#else
            fprintf(stderr, "Unable to daemonise\n");
#endif
            _dcn_receive();
            fprintf(stderr, "Something broke whilst receiving\n");
            exit(5);
#ifdef CAN_DAEMON
        }
        /*fprintf(stderr, "Successfully forked off process %d\n", (int)pid);*/
        _exit(0);
#endif
    }
    else if (mode == NONE)
    {
        fprintf(stderr, "Must specify either master or slave\n");
        exit(3);
    }
    exit(10);
#else /* !CONCURRENT */
    return call_main(argc, argv, envp);
#endif
```

```
}

#if !CONCURRENT
int _dci_doping(int ac, char *av[])
{
    pid_t p;
    while(ac)
    {
        printf("%s...", *av);
        fflush(stdout);
        if ((p = _dcn_pingn(*av)))
            printf("Up (%ld)\n", (long)p);
        else
            printf("Down\n");
        ac--, av++;
    }
    return 0;
}

int _dci_dokill(int ac, char *av[])
{
    pid_t p;
    while(ac)
    {
        printf("%s...", *av);
        fflush(stdout);
        if ((p = _dcn_pingn(*av)))
            printf("Up (%ld)\n", (long)p);
        else
            printf("Down\n");
        if (p)
        {
            _dcn_killn(*av);
//          char szBuffer[128];
//          sprintf(szBuffer, "rsh %s 'kill %ld;sleep 5;kill -9 %ld'",
    //              *av, (long)p, (long)p);
//          printf(szBuffer);
//          system(szBuffer);
        }
        ac--, av++;
    }
    return 0;
}

int _dci_pinghosts()
{
    int n = _dcn_gethostcount();
    int i;
    pid_t p;

    for(i = 0; i < n; i++)
    {
        union
        {
            struct in_addr addr;
            unsigned char b[4];
        } a;
        a.addr = _dcn_getahost(i);
        printf("%d.%d.%d.%d...",
               a.b[0], a.b[1], a.b[2], a.b[3]);
        fflush(stdout);
        if ((p = _dcn_pinga(a.addr)))
            printf("Up (%ld)\n", (long)p);
        else
            printf("Down\n");
    }
```

```
    return 0;
}

int _dci_killhosts()
{
    int n = _dcn_gethostcount();
    int i;
    pid_t p;

    for(i = 0; i < n; i++)
    {
        union
        {
            struct in_addr addr;
            unsigned char b[4];
        } a;
        a.addr = _dcn_getahost(i);
        printf("%d.%d.%d.%d...",
               a.b[0], a.b[1], a.b[2], a.b[3]);
        fflush(stdout);
        if ((p = _dcn_pinga(a.addr)))
        {
            printf("Up (%ld)...", (long)p);

            if (_dcn_killa(a.addr))
                printf("killed\n");
            else
                printf("not killed\n");
        }
        else
            printf("Down\n");
    }
    return 0;
}
#endif /* !CONCURRENT */

static int call_main(int argc, char *argv[], char *envp[])
{
    pfn fn;
    void *handle;
    TRACE1("Opening image: %s\n", _dc_ObjectName);
    handle = dlopen(_dc_ObjectName, RTLD_LAZY);

    if (!handle)
    {
        fprintf(stderr, "%s:%s\n", _dc_ObjectName, dlerror());
        return 2;
    }

    fn = (pfn)dlsym(handle, "main");

    TRACE0("Got function pointer, invoking\n");
    return fn(argc, argv, envp);
}

void set_limits(void)
{
    struct rlimit r;
    r.rlim_cur = 1023;
    r.rlim_max = 1023;

    if (setrlimit(RLIMIT_NOFILE, &r))
        perror("Setting limits");
}
```

B.8.3 dcd2/exports.c

```
/* Exported functions available to dynamic objects called by this program */
#include "project.h"
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <osthreads.h>
#include <dlfcn.h>
#include <signal.h>
#include <setjmp.h>
#include "dcerror.h"
#include "trace.h"
#include "align.h"
#include "remote.h"
#include "dist.h"

#define STAT_PREEXEC 0
#define STAT_RUNNING 1
#define STAT_FINISHED 3
#define STAT_DONE 7

static int _dci_check_status(int nSets, va_list va);

typedef int (*pfn)(struct _dch_invoke *, struct _dch_var *);

struct _dci_thread_args
{
    char *pszHost;
    struct _dch_var *pVar;
    struct _dch_invoke *pInvoke;
    int nStatus;
};

struct _dci_thread_status
{
    pthread_mutex_t lock;
    pthread_cond_t finished;
    pthread_t tid;
    unsigned int qcurrent;
};

static struct _dci_thread_status thread_status =
{
    PTHREAD_MUTEX_INITIALIZER,
    PTHREAD_COND_INITIALIZER,
    NULL,
    0U
};

static struct _dci_thread_args *package_args(const char *, struct _dch_var *,
                                             struct _dch_invoke *);

int _dci_startfn(struct _dch_invoke *pInvoke, struct _dch_var *pVar)
{
    void *handle;
    int status;
    pfn p;

    handle = dlopen(_dc_ObjectName, RTLD_LAZY);

    if (!handle)
    {
        TRACE0("Failed to get handle on object\n");
```

```
        return DE_BADOBJ;
    }

    /*TRACE1("Looking up function: \'%s\'\n", pArgs->pInvoke->szName);*/
    p = (pfn)dlsym(handle, pInvoke->szName);

    ASSERT(p != NULL);

    /*TRACE3("About to call remote function at %p with args %p, %p\n", p,
           pArgs->pInvoke, pArgs->pVar);*/
    status = p(pInvoke, pVar);

    dlclose(handle);
    return status;
}

/*
 * Thread function, used by RemoteExec to actually launch the process
 * and then wait for it complete.
 */

static void *thread_start(void *pa)
{
    struct _dci_thread_args *pArgs = (struct _dci_thread_args *)pa;

    /*TRACE1("Starting to do remote call (%ld)\n", (long)pthread_self());*/
#if 0
    if (strcmp(pArgs->pszHost, "localhost"))
    {
        pArgs->nStatus = DE_NOTIMP;
        return NULL;
    }
#endif

#if CONCURRENT
    pArgs->nStatus = _dci_startfn(pArgs->pInvoke, pArgs->pVar);
#else
    pArgs->nStatus = _dcn_send(pArgs->pszHost, pArgs->pInvoke, pArgs->pVar);
#endif

    if (pArgs->nStatus)
    {
        TRACE1("Status returned by function %d\n", pArgs->nStatus);
        abort();
    }

    TRACE1("About to signal thread (%08lX) completion\n",
           (long)pthread_self());
    pArgs->pVar->status = STAT_FINISHED;
    pthread_mutex_lock(&thread_status.lock);
    pArgs->pVar->qpos = ++thread_status.qcurrent;
    pthread_cond_broadcast(&thread_status.finished);
    pthread_mutex_unlock(&thread_status.lock);
    TRACE1("Signalled thread (%08lX) completion\n", (long)pthread_self());
    TRACE1("Finished executing remote function, status=%d\n", pArgs->nStatus);

    return pArgs;
}

int _dcx_exec(const char *pszHost, struct _dch_var *pVar,
              struct _dch_invoke *pInvoke)
{
    struct _dci_thread_args *pArgs;
    int e;

    pArgs = package_args(pszHost, pVar, pInvoke);
```

```
    if (!pArgs)
    {
        abort();
        return DE_NOMEM;
    }

    pArgs->pVar->status = STAT_PREEXEC;

    e = pthread_create(&(pArgs->pVar->tid), NULL, thread_start,
                       (void *)pArgs);
    if (e)
    {
        free(pArgs);
        abort();
        return DE_THREAD;
    }

    TRACE1("Thread created, id=%ld\n", (long)(pArgs->pVar->tid));
    return DE_SUCCESS;
}

int _dcx_select(int nSets, ...)
{
    va_list va;
    int nIndex;
    va_start(va, nSets);

    TRACE0("Starting to wait for thread set...\n");
    pthread_mutex_lock(&thread_status.lock);

    while ((nIndex = _dci_check_status(nSets, va)) < 0)
    {
        TRACE0("Waiting on cond\n");
        pthread_cond_wait(&thread_status.finished, &thread_status.lock);
        TRACE0("Cond finished\n");
    }
    TRACE0("A thread has finished\n");
    pthread_mutex_unlock(&thread_status.lock);
    return nIndex;
}

int _dci_check_status(int nSets, va_list va)
{
    int i,j;
    int nIndex = 0;
    int nChosen = -1;
    unsigned int qposChosen = (unsigned int)-1;

    for(i = 0; i < nSets; i++)
    {
        union rep
        {
            struct _dch_var **array;
            struct _dch_var *ptr;
        };

        int nSize;
        nSize = va_arg(va, int);

        /* If nSize is given as zero then pSet is not an array, */
        /* it is a single pointer */
        if (nSize)
        {
            struct _dch_var **pSet;
            pSet = va_arg(va, struct _dch_var **);
```

```
            TRACE1("Arg is an array %p\n", pSet);
            for(j = 0; j < nSize; j++)
            {
                TRACE2("Index %d into pointer %p\n", j, pSet);
                TRACE2("Checking thread %08lX, stat=%d\n", (long) pSet[j]->tid,
                       pSet[j]->status);
                if ((pSet[j]->tid != NULL) &&
                    (pSet[j]->status == STAT_FINISHED))
                {
                    TRACE2("Looking at thingy with qpos=%d, chosen level=%d\n",
                           pSet[j]->qpos, qposChosen);
                    if (qposChosen >= pSet[j]->qpos)
                    {
                        TRACE1("Picked it %d\n", nChosen);
                        nChosen = nIndex;
                        qposChosen = pSet[j]->qpos;
                    }
                }
                ++nIndex;
            }
        }
        else
        {
            struct _dch_var *pVar;
            pVar = va_arg(va, struct _dch_var *);
            TRACE1("Arg is a pointer %p\n", pVar);

            if ((pVar->tid != NULL) && (pVar->status == STAT_FINISHED))
            {
                if (qposChosen >= pVar->qpos)
                    nChosen = nIndex;
            }
            ++nIndex;
        }

    }
    return nChosen;
}

int _dcx_wait(struct _dch_var *pVar)
{
    struct _dci_thread_args *result;
    int e;

    TRACE1("Waiting for thread, id=%08lX\n", (long)(pVar->tid));

    e = pthread_join(pVar->tid, (void **) &result);

    TRACE1("Thread %08lX has finished\n", (long)(pVar->tid));

    if (e)
    {
        abort();
        return DE_THREAD;
    }

    if (!result)
    {
        abort();
        return DE_THREAD;
    }

    if (result->nStatus)
    {
        int n = result->nStatus;
        free(result);
```

```
        return n;
    }

    pVar->tid = NULL;
    pVar->status = STAT_DONE;
    TRACE0("Finished waiting\n");
    return DE_SUCCESS;
}

void _dcx_setfn(struct _dch_invoke *p, const char *s)
{
    ASSERT(strlen(s) < 64);
    strcpy(p->szName, s);
}

/*
 * Package up the arguments so they can be passed to the thread, note that
 * the host argument is copied just in case it goes out of scope.
 * (pVar is allocated by us and remains until a successful dwait so it can't
 * go out of scope. Similarly with pInvoke except the thread deletes the
 * invoke parameters when they are no longer required.)
 */
static struct _dci_thread_args *package_args(const char *pszHost,
                                             struct _dch_var *pVar,
                                             struct _dch_invoke *pInvoke)
{
    struct _dci_thread_args *pArgs;
    char *p;
    size_t cbSize;

    TRACE0("Preparing to copy arguments\n");
    if (pszHost == NULL)
        cbSize = ALIGN(sizeof(struct _dci_thread_args));
    else
        cbSize = (ALIGN(sizeof(struct _dci_thread_args)) +
                  ALIGN(strlen(pszHost) + 1));

    p = _dcx_alloc(cbSize);

    if (!p)
        return NULL;

    pArgs = (struct _dci_thread_args *)p;
    p += ALIGN(sizeof(struct _dci_thread_args));

    if (pszHost)
    {
        pArgs->pszHost = p;
        p += ALIGN(strlen(pszHost) + 1);
        strcpy(pArgs->pszHost, pszHost);
    }
    else
        pArgs->pszHost = NULL;

    /* Don't need to copy these, they will be handled correctly */
    pArgs->pVar = pVar;
    pArgs->pInvoke = pInvoke;

#ifdef DEBUG
    ASSERT(((char *)pArgs) + cbSize == p);
#endif

    pArgs->nStatus = 0;

    TRACE0("Arguments copied\n");
    return pArgs;
```

```
}

void *_dcx_alloc(size_t cbSize)
{
    void *p;
    p = malloc(cbSize);
    TRACE2("Allocated %d bytes at %p\n", cbSize, p);
    return p;
}

void *_dcx_realloc(void *p, size_t cbSize)
{
    void *q = realloc(p, cbSize);
    if (q == NULL)
        abort();
    return q;
}

void _dcx_free(void *p)
{
    TRACE1("Freeing memory at %p\n", p);
    free(p);
}

void *_dcx_allocvar(size_t cbSize)
{
    void *p = _dcx_alloc(cbSize);
    ((struct _dch_var *)p)->cbPayload = cbSize - sizeof(struct _dch_var);
    ((struct _dch_var *)p)->status = STAT_PREEXEC;
    ((struct _dch_var *)p)->tid = NULL;
    return p;
}

void *_dcx_allocargs(size_t cbSize)
{
    void *p = _dcx_alloc(cbSize);
    ((struct _dch_invoke *)p)->cbPayload = cbSize - sizeof(struct _dch_invoke);
    return p;
}
```

B.8.4 dcd2/hosts.c

```
#include "project.h"
#include "osthreads.h"
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/utsname.h>
#include <stdio.h>
#include <netdb.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "remote.h"
#include "hosts.h"

struct in_addr _dc_Hosts[HOSTS_MAX+1];
int _dc_currenthost;
int _dc_hostcount;
pthread_mutex_t hostlock = PTHREAD_MUTEX_INITIALIZER;

static int _dcn_readhosts(const char *);

int _dcn_findhosts()
{
```

```
    char szFile[PATH_MAX];

    strcpy(szFile, _dc_ObjectName);

    if (!strcmp(&szFile[strlen(szFile) - 3], ".so"))
        szFile[strlen(szFile) - 2] = '\0';
    strcat(szFile, "hosts");

    return _dcn_readhosts(szFile);
}

static int _dcn_readhosts(const char *szFile)
{
    int n = 0;
    char szBuffer[256];
    FILE *fp;
    struct hostent *he;
    struct utsname un;
    uname(&un);

    fp = fopen(szFile, "r");

    if (fp)
    {
        while (fgets(szBuffer, sizeof(szBuffer), fp) && n < HOSTS_MAX)
        {
            if (szBuffer[strlen(szBuffer) - 1] == '\n')
                szBuffer[strlen(szBuffer) - 1] = '\0';
            he = gethostbyname(szBuffer);
            if (he)
            {
#if 0
                int i;
                printf("%s...", he->h_name);
                for(i = 0; i < he->h_length; i++)
                    printf("%d.", (unsigned char)he->h_addr_list[0][i]);
                printf(" (%p)\n", he);
#endif
                ASSERT(he->h_addrtype == AF_INET);
                _dc_Hosts[n++] = *((struct in_addr *)he->h_addr_list[0]);

                if (!strncmp(szBuffer, un.nodename, strlen(un.nodename)))
                    _dc_currenthost = n;
            }
            else
                fprintf(stderr, "\nFailed to find host %s\n", szBuffer);
        }
        _dc_hostcount = n;
        if (_dc_currenthost >= _dc_hostcount)
            _dc_currenthost = 0;

        /*printf("Starting with host %d\n", _dc_currenthost);*/
        fclose(fp);

#if 0
        for(i = 0; i < _dc_hostcount; i++)
        {
            printf("%08lX\n", *((unsigned long *)(&_dc_Hosts[i])));
        }
#endif
        return 0;
    }
    else
    {
        fprintf(stderr, "Unable to open hosts file %s\n", szFile);
        return -1;
```

```
    }
}

struct in_addr _dcn_gethost()
{
    int i;
    pthread_mutex_lock(&hostlock);
    if (++_dc_currenthost >= _dc_hostcount)
        _dc_currenthost = 0;

    TRACE1("First host = %08X\n", *((unsigned int *)_dc_Hosts));
    TRACE3("This host (%d/%d) = %08X\n", _dc_currenthost,
           _dc_hostcount, *((unsigned int *)&_dc_Hosts[_dc_currenthost]));

    i = _dc_currenthost;
    pthread_mutex_unlock(&hostlock);

    ASSERT(*((int *)&_dc_Hosts[i]) != 0);
    return _dc_Hosts[i];
}

int _dcn_gethostcount()
{
    return _dc_hostcount;
}

struct in_addr  _dcn_getahost(int i)
{
    return _dc_Hosts[i];
}

int _dcn_rshslaves()
{
    union
    {
        struct in_addr addr;
        unsigned char b[4];
    } quad;

    int i;
    char szBuffer[PATH_MAX+40];
    char *display;

    display = getenv("DISPLAY");
    if (display == NULL)
        display = "";

    for(i = 0; i < _dc_hostcount; i++)
    {
        quad.addr = _dc_Hosts[i];
        sprintf(szBuffer, "rsh %d.%d.%d.%d `which dcd` -s %s &",
                quad.b[0], quad.b[1], quad.b[2], quad.b[3],
                _dc_ObjectName);
        printf("%s\n", szBuffer);
        system(szBuffer);
    }
    return 0;
}
```

B.8.5 dcd2/dist.c

```
#include "project.h"
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
```

```
#include <netdb.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>

#include "sockutil.h"
#include "remote.h"
#include "dist.h"
#include "hosts.h"

#if !CONCURRENT
int swrite(int fd, void *buf, int n)
{
    return (xwrite(fd, buf, n) == n);
}

int sread(int fd, void *buf, int n)
{
    return (xread(fd, buf, n) == n);
}

void *_dci_acceptproc(void *);

static int _dci_openoutgoing(struct in_addr addr)
{
    union
    {
        struct in_addr addr;
        unsigned char b[4];
    } boing;

    int fd;
    struct sockaddr_in name;

    boing.addr = addr;
    name.sin_family = AF_INET;
    name.sin_addr = addr;
    name.sin_port = htons(DCPORT);

    /*TRACE4("Opening to %d.%d.%d.%d\n",
           boing.b[0], boing.b[1], boing.b[2], boing.b[3]);*/

    fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd < 0)
    {
        perror("Opening socket");
        return -1;
    }
    if (connect(fd, (struct sockaddr *) &name, sizeof(name)) < 0)
    {
        /*perror("Connecting to socket");*/
        close(fd);
        return -1;
    }

    return fd;
}

int _dci_openincoming()
{
    int fd;
    struct sockaddr_in name;
    int length;
    int option = 1;
```

```
    fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd < 0)
    {
        perror("Opening socket");
        return -1;
    }

    name.sin_family = AF_INET;
    name.sin_addr.s_addr = INADDR_ANY;
    name.sin_port = htons(DCPORT);
    setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, (char *)&option, sizeof(option));
    if (bind(fd, (struct sockaddr *) &name, sizeof(name)) < 0)
    {
        perror("binding socket.");
        exit(1);
    }
    length = sizeof(name);
    if (getsockname(fd, (struct sockaddr *) &name, &length) < 0)
    {
        perror("Error getting socket name");
        exit(1);
    }

    listen(fd, 5);
    return fd;
}

int _dci_acceptconnection(int fdSocket)
{
    int fd;
    fd = accept(fdSocket, (struct sockaddr *) 0, (int *) 0);
    TRACE1("Accepted connection as %d\n", fd);
    makeblock(fd);
    return fd;
}

pid_t _dcn_pingn(const char *szHost)
{
    struct hostent *phe;
    struct in_addr addr;

    phe = gethostbyname(szHost);
    if (!phe)
        return 0;

    addr = *((struct in_addr *)phe->h_addr_list[0]);

    return _dcn_pinga(addr);
}

pid_t _dcn_pinga(struct in_addr addr)
{
    int fd = -1;
    int data;
    pid_t pid;

    fd = _dci_openoutgoing(addr);
    if (fd < 0)
        return 0;

    data = DCPING;
    if (!swrite(fd, &data, sizeof(data)))
    {
        close(fd);
        return 0;
```

```
    }
    if (!sread(fd, &data, sizeof(data)))
    {
        close(fd);
        return 0;
    }
    pid = (pid_t)data;
    if (!sread(fd, &data, sizeof(data)) || data != DCPONG)
    {
        close(fd);
        return 0;
    }

    close(fd);
    return pid;
}

int _dcn_killn(const char *szHost)
{
    struct hostent *phe;
    struct in_addr addr;

    phe = gethostbyname(szHost);
    if (!phe)
        return 0;

    addr = *((struct in_addr *)phe->h_addr_list[0]);

    return _dcn_killa(addr);
}

int _dcn_killa(struct in_addr addr)
{
    int fd = -1;
    int data;

    fd = _dci_openoutgoing(addr);
    if (fd < 0)
        return 0;

    data = DCKILL;
    if (!swrite(fd, &data, sizeof(data)))
    {
        close(fd);
        return 0;
    }
    if (!sread(fd, &data, sizeof(data)) || data != DCPONG)
    {
        close(fd);
        return 0;
    }

    close(fd);
    return 1;
}

int _dcn_send(const char *szHost, struct _dch_invoke *pi, struct _dch_var *pv)
{
    int fd = -1;
    int data;
    int status;
    struct in_addr addr;

    if (szHost)
    {
        struct hostent *pHost;
```

```
        pHost = gethostbyname(szHost);
        if (pHost == NULL)
        {
            herror("Getting host name");
            return -1;
        }
        addr = *((struct in_addr *)pHost->h_addr_list[0]);
        fd = _dci_openoutgoing(addr);
#ifdef DEBUG
        if (fd < 0)
            TRACE1("Unable to connect to specified host %s, trying others\n",
                   szHost);
#endif
    }

    while(fd < 0)
    {
        addr = _dcn_gethost();
        TRACE1("Going to send message to %08lX\n",
               *((unsigned long *)(&addr)));
        fd = _dci_openoutgoing(addr);
    }

    TRACE0("Sending preamble\n");
    /* Send */
    data = DCPREAMBLE;
    if (!swrite(fd, &data, sizeof(data)))
    {
        close(fd);
        return -1;
    }

    TRACE0("Sending invoke structure\n");
    data = sizeof(struct _dch_invoke) + pi->cbPayload;
    if (xwrite(fd, &data, sizeof(data)) != sizeof(data))
    {
        close(fd);
        return -1;
    }
    if (!swrite(fd, pi, sizeof(struct _dch_invoke) + pi->cbPayload))
    {
        close(fd);
        return -1;
    }

    TRACE0("Sending result structure\n");
    data = sizeof(struct _dch_var) + pv->cbPayload;
    if (!swrite(fd, &data, sizeof(data)))
    {
        close(fd);
        return -1;
    }
#if 0
    if (!swrite(fd, pv, sizeof(struct _dch_var) + pv->cbPayload))
    {
        close(fd);
        return -1;
    }
#else
    /* Just send the header */
    if (!swrite(fd, pv, sizeof(struct _dch_var)))
    {
        close(fd);
        return -1;
    }
#endif
```

```
    TRACE0("Sending postamble\n");
    data = DCPOSTAMBLE;
    if (!swrite(fd, &data, sizeof(data)))
    {
        close(fd);
        return -1;
    }

    TRACE0("Waiting for result\n");
    /* Receive */
    if (!sread(fd, &data, sizeof(data)) || data != DCPREAMBLE)
    {
        close(fd);
        return -1;
    }
    TRACE0("Preamble read\n");

    if (!sread(fd, &status, sizeof(status)))
    {
        close(fd);
        return -1;
    }
    if (status == 0)
    {
        if (!sread(fd, &data, sizeof(data)))
        {
            close(fd);
            return -1;
        }
        if (data != sizeof(struct _dch_var) + pv->cbPayload)
        {
            TRACE0("Dodgy data length retrieved\n");
            close(fd);
            return -1;
        }
        if (!sread(fd, pv, sizeof(struct _dch_var) + pv->cbPayload))
        {
            close(fd);
            return -1;
        }
        TRACE0("Result read\n");
    }

    if (!sread(fd, &data, sizeof(data)) || data != DCPOSTAMBLE)
    {
        close(fd);
        return -1;
    }
    TRACE0("Postamble read\n");

    close(fd);
    return status;
}

int _dcn_receive()
{
    int fdSocket, fdConnection;

    TRACE0("Opening incoming socket\n");
    fdSocket = _dci_openincoming();
    TRACE1("Opened incoming socket as %d\n", fdSocket);

    if (fdSocket < 0)
        return -1;

    while (1)
```

```
    {
        pthread_t tid;
        fdConnection = _dci_acceptconnection(fdSocket);

        pthread_create(&tid, NULL, _dci_acceptproc, (void *)fdConnection);
    }
    return 0;
}

void *_dci_acceptproc(void *p)
{
    int fd = (int) p;
    int data;
    int status;
    struct _dch_invoke *pi;
    struct _dch_var *pv;

    TRACE0("Accept thread started\n");

    /* Receive */
    if (!sread(fd, &data, sizeof(data)) || data != DCPREAMBLE)
    {
        if (data == DCPING)
        {
            TRACE0("Got a ping\n");
            data = (int)_dc_pid;
            swrite(fd, &data, sizeof(data));
            data = DCPONG;
            swrite(fd, &data, sizeof(data));
            close(fd);
            return NULL;
        }
        else if (data == DCKILL)
        {
            TRACE0("Got a kill request\n");
            data = DCPONG;
            swrite(fd, &data, sizeof(data));
            close(fd);
            kill(_dc_pid, SIGTERM);
            return NULL;
        }
        else
        {
            TRACE0("Failed to get preamble, closing\n");
            close(fd);
            return (void *)-1;
        }
    }
    TRACE0("Got preamble\n");

    if (!sread(fd, &data, sizeof(data)))
    {
        close(fd);
        return (void *)-1;
    }
    TRACE1("Got invoke length %d\n", data);
    pi = _dcx_alloc(data);
    if (!sread(fd, pi, data))
    {
        close(fd);
        return (void *)-1;
    }
    TRACE1("Got invoke data, function=%s\n", pi->szName);

    if (!sread(fd, &data, sizeof(data)))
    {
```

```
        close(fd);
        return (void *)-1;
    }
    TRACE1("Got return length %d\n", data);
    pv = _dcx_alloc(data);
#if 0
    if (!sread(fd, pv, data))
    {
        close(fd);
        return (void *)-1;
    }
#else
    if (!sread(fd, pv, sizeof(struct _dch_var)))
    {
        close(fd);
        return (void *)-1;
    }
#endif
    TRACE0("Got return data\n");

    if (!sread(fd, &data, sizeof(data)) || data != DCPOSTAMBLE)
    {
        close(fd);
        return (void *)-1;
    }

    status = _dci_startfn(pi, pv);

    TRACE0("Function executed, returning result\n");
    /* Send */
    data = DCPREAMBLE;
    if (!swrite(fd, &data, sizeof(data)))
    {
        close(fd);
        return (void *)-2;
    }

    if (!swrite(fd, &status, sizeof(status)))
    {
        close(fd);
        return (void *)-2;
    }

    data = sizeof(struct _dch_var) + pv->cbPayload;
    if (!swrite(fd, &data, sizeof(data)))
    {
        close(fd);
        return (void *)-2;
    }
    if (!swrite(fd, pv, data))
    {
        close(fd);
        return (void *)-2;
    }

    data = DCPOSTAMBLE;
    if (!swrite(fd, &data, sizeof(data)))
    {
        close(fd);
        return (void *)-2;
    }

    TRACE0("Result returned, closing connection\n");
    close(fd);
    return NULL;
}
```

```
#endif
```

B.8.6 dcd2/bucket.c

```
#include "project.h"
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <osthreads.h>
#include <dlfcn.h>
#include <signal.h>
#include <setjmp.h>
#include "dcerror.h"
#include "trace.h"
#include "align.h"
#include "remote.h"
#include "dist.h"

#define BUCKETSIZE 10

struct bnode
{
    void *pResult;
    struct _dch_var *pVar;
};

struct bucket
{
    int nIndex;
    struct bnode *pNodes;
    size_t nBucketSize;
};

_dct_bucket _dcx_bucket_create()
{
    struct bucket *pb = _dcx_alloc(sizeof(struct bucket));
    pb->nBucketSize = BUCKETSIZE;
    pb->pNodes = _dcx_alloc(sizeof(struct bnode) * pb->nBucketSize);
    pb->nIndex = 0;

    TRACE1("Bucket created at %p\n", pb);
    return pb;
}

void _dcx_bucket_add(_dct_bucket b, void *r, struct _dch_var *pv)
{
    struct bucket *pb = (struct bucket *)b;
    ASSERT(pb->nIndex <= pb->nBucketSize);
    if (pb->nIndex == pb->nBucketSize)
    {
        pb->nBucketSize *= 2;
        TRACE1("Enlarging bucket to size %d\n", pb->nBucketSize);
        pb->pNodes = _dcx_realloc(pb->pNodes,
                                  sizeof(struct bnode) * pb->nBucketSize);
    }
    pb->pNodes[pb->nIndex].pResult = r;
    pb->pNodes[pb->nIndex].pVar = pv;
    TRACE3("Bucket node added at %d, result at %p, var at %p\n", pb->nIndex,
           r, pv);
    pb->nIndex++;
}

void _dcx_bucket_wait(_dct_bucket b)
{
    struct bucket *pb = (struct bucket *)b;
```

```
    int i;
    TRACE1("Doing wait on bucket at %p\n", pb);
    for(i = 0; i < pb->nIndex; i++)
    {
        int nLength;
        TRACE2("Waiting for %d at %p\n", i, pb->pNodes[i].pVar);
        _dcx_wait(pb->pNodes[i].pVar);
        nLength = pb->pNodes[i].pVar->cbPayload;
        TRACE1("Got an answer, copying - length=%d\n", nLength);
        memcpy(pb->pNodes[i].pResult, pb->pNodes[i].pVar + 1, nLength);
    }
    _dcx_free(pb);
}
```

B.8.7 dcd2/dist.h

```
#ifndef DIST_H
#define DIST_H 1

#include "remote.h"
#include <netinet/in.h>

int _dcn_receive();
int _dcn_send(const char *szHost, struct _dch_invoke *pi, struct _dch_var *pv);
pid_t _dcn_pingn(const char *szHost);
int _dcn_killn(const char *szHost);
pid_t _dcn_pinga(struct in_addr addr);
int _dcn_killa(struct in_addr addr);

#endif
```

B.8.8 dcd2/hosts.h

```
#ifndef HOSTS_H
#define HOSTS_H 1

#define HOSTS_MAX 32

#include <netinet/in.h>

extern int _dcn_findhosts();
extern int _dcn_rshslaves();
extern int _dcn_gethostcount();
extern struct in_addr _dcn_getahost(int);
extern struct in_addr _dcn_gethost();

#endif
```

B.9 Library

B.9.1 lib/Makefile.in

```
# Makefile.in for project/lib

srcdir = @srcdir@
VPATH = @srcdir@
TOP = ..
THIS= lib
```

```
SUBDIRS=src test include

include $(TOP)/Make.rules

all : bin

bin :
	for i in $(SUBDIRS); do $(MAKE) -C $$i bin; done

dep :
	for i in $(SUBDIRS); do $(MAKE) -C $$i dep; done

doc : doc_me
	for i in $(SUBDIRS); do $(MAKE) -C $$i doc; done

clean : clean_me
	for i in $(SUBDIRS); do $(MAKE) -C $$i clean; done

distclean : clean_me
	for i in $(SUBDIRS); do $(MAKE) -C $$i distclean; done
	$(RM) Makefile

install :

clean_me :
	$(RM) *

doc_me : pretty.tex

pretty.tex : Makefile.in
	$(TEXIFY) $^ > pretty.tex

Makefile : Makefile.in $(TOP)/config.status
	cd $(TOP) && CONFIG_FILES=$(THIS)/Makefile CONFIG_HEADERS= ./config.status
```

B.9.2 lib/include/Makefile.in

```
# Makefile.in for project/lib/include

srcdir = @srcdir@
VPATH = @srcdir@
TOP = ../..
THIS=lib/include

HEADERS = project.h sockutil.h mstring.h trace.h dcerror.h \
      objhash.h remote.h align.h dnew.h osthreads.h

include $(TOP)/Make.rules

all : bin

bin :

clean :
	$(RM) *  pretty.tex

dep :

doc : pretty.tex

pretty.tex : Makefile.in $(HEADERS)
	$(TEXIFY) $^ > pretty.tex

distclean : clean
```

```
	$(RM) Makefile config.h

Makefile : Makefile.in $(TOP)/config.status
	cd $(TOP) && CONFIG_FILES=$(THIS)/Makefile CONFIG_HEADERS= ./config.status
```

B.9.3 lib/include/project.h

```
#ifndef PROJECT_H
#define PROJECT_H 1
#include <config.h>

#ifdef HAVE_STDLIB_H
#include <stdlib.h>
#endif

#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif

#ifdef HAVE_LIMITS_H
#include <limits.h>
#endif

// Use debugging new/delete
#ifdef __cplusplus
#include <new>
#include <dnew.h>
#endif

#include <trace.h>

#ifndef PATH_MAX
#ifdef MAXPATHLEN
#define PATH_MAX MAXPATHLEN
#else
#error No PATH_MAX or MAXPATHLEN
#endif
#endif

#if HAVE_SYS_UTSNAME_H
#if HAVE_SETSID
#if HAVE_FORK
#if HAVE_DUP
#define CAN_DAEMON 1
#endif
#endif
#endif
#endif

#endif
```

B.9.4 lib/include/sockutil.h

```
#ifndef SOCKUTIL_H
#define SOCKUTIL_H 1

int xread(int fd, void *buf, int n);
int xwrite(int fd, void *buf, int n);
void ignore_sigpipe(int ign);
void makeblock(int fd);

#endif
```

B.9.5 lib/include/mstring.h

```
/* MString is a general string class, used throughout the program. */

#ifndef MSTRING_H_
#define MSTRING_H_

#include <string.h>

class MString
{
protected:
    char *m_pszString;
    int m_nSpace;

    void MakeSpace(int nSize);
    void Fill(const char *pszString, int nCount = -1);
    void Fill(const MString &str, int nCount = -1);

    MString(int nSize);
 public:
    MString();
    MString(const char *pszString, int nCount = -1);
    MString(const MString &str, int nCount = -1);
    MString(char cChar);

    MString &operator=(const char *pszString);
    MString &operator=(const MString &str);
     MString();

    void Empty();

    MString &operator+=(const MString &str);
    friend MString operator+(const MString &str1, const MString &str2);

    char *Grab(int nSize = -1);
    void Release(int nSize = -1);

    operator const char *() const;
    int GetLength() const;

    friend bool operator==(const MString &str1, const MString &str2);
    friend bool operator==(const MString &str1, const char *sz);
    friend bool operator==(const char *sz, const MString &str2);
    friend bool operator<(const MString &s1, const MString &s2);
    friend bool operator>(const MString &s1, const MString &s2);

    void DropLeft(int nCount = 1);
    void DropRight(int nCount = 1);

    void Insert(int nPos, char ch);
    void Remove(int nPos);

    MString Left(int nCount = 1) const;
    MString Right(int nCount = 1) const;
    MString After(int nStart) const;
    MString Mid(int nStart, int nLength) const;
    void LTrim();
    void RTrim();

    int Find(char ch, int nStart = 0) const;
    int Find(const char *pszSearch, int nStart = 0) const;
```

```
    char Last() const
    {
        int l;
        if (m_pszString && (l = strlen(m_pszString)))
            return m_pszString[strlen(m_pszString) - 1];
        else
            return '\0';
    }

    char First() const
    {
        if (m_pszString)
            return *m_pszString;
        else
            return '\0';
    }

    char operator[](int nIndex) const
    {
        if(!m_pszString)
            return 0;
        else
            return m_pszString[nIndex];
    }

    char &operator[](int nIndex)
    {
        if(!m_pszString)
            MakeSpace(1);
        return m_pszString[nIndex];
    }
};

inline bool operator<(const MString &s1, const MString &s2)
{
    if ((s1.m_pszString == NULL) || (s2.m_pszString == NULL))
        return (s1.m_pszString == NULL) && (s2.m_pszString != NULL);
    else
        return strcmp(s1.m_pszString, s2.m_pszString) < 0;
}

inline bool operator>(const MString &s1, const MString &s2)
{
    if ((s1.m_pszString == NULL) || (s2.m_pszString == NULL))
        return (s1.m_pszString != NULL) && (s2.m_pszString == NULL);
    else
        return strcmp(s1.m_pszString, s2.m_pszString) > 0;
}

#endif
```

B.9.6 lib/include/trace.h

```
#ifndef TRACE_H
#define TRACE_H 1

#include <stdio.h>
#include <stdarg.h>
#include <unistd.h>

#ifdef DEBUG

#ifdef __cplusplus

// The C++ solution
```

```
class CTrace
{
    FILE *m_fp;
public:
    CTrace();
     CTrace();

    void Out(const char *fmt, va_list v);
    static void Trace(const char *fmt, ...) __attribute__((format(printf, 1, 2)));
};

extern "C"
{
    void _dc_Trace(const char *, ...) __attribute__((format(printf, 1, 2)));
}

#define TRACEX(A) CTrace::Trace A
#else

void _dc_Trace(const char *, ...) __attribute__((format(printf, 1, 2)));

#define TRACEX(A) _dc_Trace A
#endif

#define TFMT "%-10s %3d : "

#define STRIZE(A) #A
#define ASSERT(A) do { if (!(A)) {TRACE1("Assertion failed '%s'\n", \
                                          STRIZE(A)); abort();}} while(0)
#define TRACE0(A) TRACEX((TFMT A, __FILE__, __LINE__))
#define TRACE1(A, B) TRACEX((TFMT A, __FILE__, __LINE__, B))
#define TRACE2(A, B, C) TRACEX((TFMT A, __FILE__, __LINE__, B, C))
#define TRACE3(A, B, C, D) TRACEX((TFMT A, __FILE__, __LINE__, B, C, D))
#define TRACE4(A, B, C, D, E) TRACEX((TFMT A, __FILE__, __LINE__, B, C, D, E))
#else
#define ASSERT(A)
#define TRACE0(A)
#define TRACE1(A,B)
#define TRACE2(A,B,C)
#define TRACE3(A,B,C,D)
#define TRACE4(A,B,C,D,E)
#define TRACEX(A)
#endif

#endif
```

B.9.7 lib/include/dcerror.h

```
/* Error codes */

#ifndef DCERROR_H
#define DCERROR_H 1

#define DE_SUCCESS 0 /* All is hunky dory */
#define DE_NOTIMP -1 /* Not implemented */
#define DE_BADOBJ -2 /* Couldn't open shared object file */
#define DE_NOMEM  -3 /* Out of memory */
#define DE_THREAD -4 /* Threading function returned error */

#endif
```

B.9.8 lib/include/objhash.h

```
#ifndef OBJHASH_H
#define OBJHASH_H 1

#ifndef new
#define OBJHASH_new_defined
#define new DEBUG_NEW
#endif

template<class OBJ, class AKEY>
class ObjectHashIterator;

template<class OBJ, class AKEY>
class ObjectHash
{
protected:
    class HashItem
    {
    public:
        HashItem *pNext;
        HashItem *pReplaced;
        OBJ obj;

        HashItem(const OBJ &o, HashItem *p, HashItem *pr)
            : pNext(p), pReplaced(pr), obj(o) {}
    };

    HashItem **pTable;
    int nTableSize;
    mutable ObjectHashIterator<OBJ, AKEY> *pIterator;

    void NotifyIterator(HashItem *p, int i)
    {
        InternalNotifyIterator(p, i);
    }

    void InternalNotifyIterator(void *, int);

    void CreateTable()
    {
//      TRACE1("Creating hash table of size %d\n", nTableSize);
        pTable = new HashItem *[nTableSize];

        HashItem **p = pTable;
        for(int i = 0; i < nTableSize; i++)
            *p++ = NULL;
    }


    void Empty()
    {
//      TRACE0("About to empty table\n");
        HashItem **p = pTable;
        for(int i = 0; i < nTableSize; i++)
        {
            while(RemoveFirstItem(p))
                ;
            p++;
        }
//      TRACE0("Finisheed emptying table\n");
    }

    bool RemoveFirstItem(HashItem **pEntry)
    {
        if (*pEntry)
        {
```

```
            HashItem *pItem = *pEntry;
//          TRACE0("Removing item\n");
            *pEntry = pItem->pNext;
            delete pItem;
            return true;
        }
        return false;
    }

    HashItem **FindEntry(AKEY key)
    {
#ifdef DEBUG
        int offset = (OBJ::HashFunction(key) % nTableSize);
//      TRACE1("Putting item in slot %d\n", offset);
        return pTable + offset;
#else
        return pTable + (OBJ::HashFunction(key) % nTableSize);
#endif
    }

    bool InsertItem(HashItem **pEntry, const OBJ &obj)
    {
        HashItem *pThis = new HashItem(obj, *pEntry, NULL);
        *pEntry = pThis;

//      TRACE2("New item with key \'%s\' inserted at %p\n",
//             obj.GetKey(), pThis);
        return true;
    }

    bool ReplaceItem(HashItem **pEntry, const OBJ &obj)
    {
        // Iterate through looking for an item with the same key.
        HashItem *pPrevious = NULL;
        HashItem *pReplace = FindItem(pEntry, obj.GetKey(), &pPrevious);

        if (pReplace)
        {
            // We found it.
            TRACE0("Found an item to replace in ReplaceItem\n");

            HashItem *pThis = new HashItem(obj, pReplace->pNext, pReplace);
            if (pPrevious)
                pPrevious->pNext = pThis;
            else
                *pEntry = pThis;
            pReplace->pNext = NULL; // Safety measure.
        }
        else
        {
            TRACE0("No item to replace in ReplaceItem\n");
            HashItem *pThis = new HashItem(obj, *pEntry, NULL);
            *pEntry = pThis;
        }
        return true;
    }

    HashItem *FindItem(HashItem **pEntry, AKEY key,
                       HashItem **ppPrevious = NULL)
    {
        HashItem *pCurrent = *pEntry;
        HashItem *pPrevious = NULL;
        while (pCurrent)
        {
            if (pCurrent->obj.IsKeyMatch(key))
                break;
```

```
            pPrevious = pCurrent;
            pCurrent = pCurrent->pNext;
        }
        if (ppPrevious)
            *ppPrevious = pPrevious;
        return pCurrent;
    }

public:
    ObjectHash(int n = 17)
    {
        pTable = NULL;
        nTableSize = n;
        pIterator = NULL;
        CreateTable();
    }

     ObjectHash()
    {
        //TRACE0("Deleting hash table\n");
        Empty();
        delete pTable;
    }

    bool Add(const OBJ &obj)
    {
        return InsertItem(FindEntry(obj.GetKey()), obj);
    }

    bool AddOrReplace(const OBJ &obj)
    {
        return ReplaceItem(FindEntry(obj.GetKey()), obj);
    }

    bool Lookup(AKEY k, OBJ &ro)
    {
        HashItem *pItem = FindItem(FindEntry(k), k);
        if (pItem)
        {
            ro = pItem->obj;
            return true;
        }
        return false;
    }

    OBJ *Lookup(AKEY k)
    {
        HashItem *pItem = FindItem(FindEntry(k), k);
        if (pItem)
            return &(pItem->obj);
        return NULL;
    }

    OBJ &operator[](AKEY k)
    {
        HashItem *pItem = FindItem(FindEntry(k), k);
        if (pItem)
            return pItem->obj;
        ASSERT(false);
        return (OBJ &)(*((void *)0));
    }

    OBJ &operator[](const ObjectHashIterator<OBJ, AKEY> &rIter);

    bool IsMember(AKEY k)
    {
```

```
        return (FindItem(FindEntry(k), k) != NULL);
    }

    void SafeRemove(HashItem **pUpdate, HashItem *pItem)
    {
        NotifyIterator(pItem, 0);
        if (pItem->pReplaced)
        {
            pItem->pReplaced->pNext = pItem->pNext;
            *pUpdate = pItem->pReplaced;
        }
        else
        {
            *pUpdate = pItem->pNext;
        }
        delete pItem;
    }

    bool Remove(AKEY k)
    {
        HashItem **pEntry = FindEntry(k);
        HashItem *pPrevious;
        HashItem *pItem = FindItem(pEntry, k, &pPrevious);

        if (pPrevious == NULL)
        {
            SafeRemove(pEntry, pItem);
            return true;
        }
        else
        {
            SafeRemove(&(pPrevious->pNext), pItem);
            return true;
        }
        return false;
    }
    bool Remove(const ObjectHashIterator<OBJ, AKEY> &rIter)
    {
        SafeRemove(rIter.pCurrentEntry, rIter.pCurrentItem);
        return true;
    }

    friend ObjectHashIterator<OBJ, AKEY>;
};

template <class OBJ, class AKEY>
class ObjectHashIterator
{
    typedef ObjectHash<OBJ, AKEY>::HashItem HashItem;
    const ObjectHash<OBJ, AKEY> &TheHash;

    HashItem **pCurrentEntry;
    HashItem *pCurrentItem;
    ObjectHashIterator<OBJ, AKEY> *pNextIterator;
    bool bSkipNextForwards;

    void FindNextEntry()
    {
        //TRACE0("Entering FindNextEntry\n");
        if (pCurrentItem)
        {
//          TRACE0("We have a current item\n");
            if (pCurrentItem->pNext)
            {
                //TRACE0("We can simply move to the next one\n");
                pCurrentItem = pCurrentItem->pNext;
```

```
                return;
            }
            //TRACE0("There are no more, let's continue\n");
            pCurrentItem = NULL;
        }
        HashItem **pBeyond = TheHash.pTable + TheHash.nTableSize;
        //TRACE2("Currently at %8p, beyond is %8p\n", pCurrentEntry, pBeyond);

        if (pCurrentEntry)
            pCurrentEntry++;
        else
            pCurrentEntry = TheHash.pTable;

        while (pCurrentEntry < pBeyond && *pCurrentEntry == NULL)
        {
            //TRACE1("In loop pCurrentEntry = %8p\n", pCurrentEntry);
            pCurrentEntry++;
        }
        if (pCurrentEntry < pBeyond)
        {
            //TRACE0("Found a non-NULL entry\n");
            pCurrentItem = *pCurrentEntry;
        }
        else
        {
            //TRACE0("Couldn't find any more entries\n");
            pCurrentItem = NULL, pCurrentEntry = NULL;
        }
    }

#ifdef DEBUG

    void DebugReplacements(HashItem *p)
    {
        if (p->pReplaced)
        {
            fprintf(stderr, "[");
            HashItem *q = p->pReplaced;
            while(q)
            {
//              fprintf(stderr, " %8p (%10s)", q, q->obj.GetKey());
                q = q->pReplaced;
            }
            fprintf(stderr, " ]");
        }
    }

    void Debug()
    {
        return;
        fprintf(stderr, "Debugging table");
        for(int i = 0; i < TheHash.nTableSize; i++)
        {
            HashItem *p = TheHash.pTable[i];
            if (!p)
                fprintf(stderr, "\n%04d NULL", i);
            else
            {
//              fprintf(stderr, "\n%04d %8p (%10s)", i, p, p->obj.GetKey());
                DebugReplacements(p);
                p = p->pNext;
                while (p)
                {
//                  fprintf(stderr, " -> %8p (%10s)", p, p->obj.GetKey());
                    DebugReplacements(p);
```

```
                    p = p->pNext;
                }
            }
            fprintf(stderr, "\n");
        }

    }
#endif

    void GotNotify(HashItem *pItem, int n)
    {
        // Currently only used for deletions so....
        if (pCurrentItem == pItem)
        {
            FindNextEntry();
            bSkipNextForwards = true;
        }
        if (pNextIterator)
            pNextIterator->GotNotify(pItem, n);
    }

public:
    ObjectHashIterator(const ObjectHash<OBJ, AKEY> &h)
        : TheHash(h)
    {
        pCurrentEntry = NULL;
        pCurrentItem = NULL;
        bSkipNextForwards = false;
        pNextIterator = TheHash.pIterator;
        if (pNextIterator)
            TRACE1("Warning: Another iterator in our midst\n", pNextIterator);
        TheHash.pIterator = this;

#ifdef DEBUG
        Debug();
#endif
        FindNextEntry();
    }

     ObjectHashIterator()
    {
        TheHash.pIterator = pNextIterator;
    }

    OBJ &GetCurrent() const
    {
        ASSERT(pCurrentEntry);
        ASSERT(pCurrentItem);
        return pCurrentItem->obj;
    }

    void operator++(int)
    {
        if (!bSkipNextForwards)
            FindNextEntry();
        bSkipNextForwards = false;
    }

    operator bool()
    {
//      TRACE2("in op bool pCurrentEntry=%08X, pCurrentItem=%08X\n",
//             pCurrentEntry, pCurrentItem);
        return !(pCurrentEntry == NULL && pCurrentItem == NULL);
    }
    friend ObjectHash<OBJ, AKEY>;
};
```

```
template<class OBJ, class AKEY>
inline OBJ &
ObjectHash<OBJ, AKEY>::operator[](const ObjectHashIterator<OBJ, AKEY> &rIter)
{
    return rIter.GetCurrent();
}

template<class OBJ, class AKEY>
inline void ObjectHash<OBJ, AKEY>::InternalNotifyIterator(void *pItem, int n)
{
    if (pIterator)
        pIterator->GotNotify((ObjectHash<OBJ, AKEY>::HashItem *)pItem, n);
}

#ifdef OBJHASH_new_defined
#undef new
#endif

#endif
```

B.9.9 lib/include/remote.h

```
#ifndef REMOTE_H
#define REMOTE_H 1

#include "osthreads.h"

#define MAX_SYMBOL_LENGTH 64

struct _dch_invoke
{
    unsigned long cbPayload;
    char szName[MAX_SYMBOL_LENGTH];
};

struct _dch_var
{
    unsigned long cbPayload;
    pthread_t tid;
    int status;
    unsigned int qpos;
};

typedef void *_dct_bucket;

extern int _dcx_exec(const char *, struct _dch_var *, struct _dch_invoke *);
extern int _dcx_wait(struct _dch_var *);
extern void _dcx_setfn(struct _dch_invoke *, const char *);
extern int _dcx_select(int, ...);

extern _dct_bucket _dcx_bucket_create();
extern void _dcx_bucket_add(_dct_bucket, void *, struct _dch_var *);
extern void _dcx_bucket_wait(_dct_bucket);

extern void *_dcx_allocvar(size_t);
extern void *_dcx_allocargs(size_t);
extern void *_dcx_alloc(size_t);
extern void _dcx_free(void *);
extern void *_dcx_realloc(void *, size_t);
extern int _dci_startfn(struct _dch_invoke *pInvoke, struct _dch_var *pVar);
extern char _dc_ObjectName[];
extern pid_t _dc_pid;
```

```
#define DCPORT 23454
#define DCPREAMBLE 0x00DCDC00
#define DCPOSTAMBLE 0xDC0000CD
#define DCPING 0x01DCDC10
#define DCPONG 0xDC0110DC
#define DCKILL 0xDCDEADDC

#endif
```

B.9.10 lib/include/align.h

```
#ifndef ALIGN_H
#define ALIGN_H 1

#define ALIGNMENT 8
#define ALIGN(N) ((N + (ALIGNMENT-1)) &  (ALIGNMENT-1))
#endif
```

B.9.11 lib/include/dnew.h

```
#ifndef DNEW_H
#define DNEW_H 1

#include <stdlib.h>

#ifdef DEBUG
void *operator new(size_t s);
void *operator new(size_t, const char *, int);
void operator delete(void *);
void *operator new[](size_t s);
void *operator new[](size_t, const char *, int);
void operator delete[](void *);
void dumpleaks();

#define DEBUG_NEW new(__FILE__, __LINE__)
#else
#define DEBUG_NEW new
#endif

#endif
```

B.9.12 lib/include/osthreads.h

```
/* Depending on what we're running on find some threads */
#ifndef OSTHREADS_H
#define OSTHREADS_H 1

#define _MIT_POSIX_THREADS 1

#include <config.h>
#ifdef HAVE_PTHREAD_H
#include <pthread.h>
#endif

#endif
```

B.9.13 lib/src/Makefile.in

```
# Makefile.in for project/lib/src

srcdir = @srcdir@
VPATH = @srcdir@
TOP = ../..
THIS=lib/src

include $(TOP)/Make.rules

SRC=trace.C mstring.C new.C sockutil.c
OBJ2=$(SRC:.C=.o)
OBJ=$(OBJ2:.c=.o)

LIB=$(LIBDIR)/libproject.a

CFLAGS += $(CFLAGS_THREAD) $(CFLAGS_SHARED)
CXXFLAGS += $(CXXFLAGS_THREAD) $(CXXFLAGS_SHARED)
LDFLAGS += (LDFLAGS_THREAD) $(LDFLAGS_SHARED)

all : dep bin

bin : $(LIB)

$(LIB) : $(OBJ)
    $(RM) $(LIB)
    ar rc $(LIB) $(OBJ)

clean :
    $(RM) $(OBJ) $(LIB) .depend *  pretty.tex

dep : .depend

.depend :
    $(DEP) $(SRC) > .depend

doc : pretty.tex

pretty.tex : Makefile.in $(SRC)
    $(TEXIFY) $^ > pretty.tex

distclean : clean
    $(RM) Makefile

Makefile : Makefile.in $(TOP)/config.status
    cd $(TOP) && CONFIG_FILES=$(THIS)/Makefile CONFIG_HEADERS= ./config.status

-include .depend
```

B.9.14 lib/src/trace.C

```
#include "project.h"
#include <stdio.h>
#include <stdarg.h>
#include "trace.h"
#include "mstring.h"

#ifdef DEBUG
#define DEVICE "/dev/tty"

CTrace trace;

bool bUseColour = false;
```

```
CTrace::CTrace()
{
    bUseColour = false;

//  m_fp = fopen(DEVICE, "w+");
    m_fp = stderr;
    if (m_fp == NULL)
        fprintf(stderr, "Couldn't open " DEVICE "\n");

// Avoid warning since Solaris pid_t is long and Linux pid_t is int.
    fprintf(m_fp, "**** Debugging started of process "
            "%ld\n", (long)getpid());
};

CTrace:: CTrace()
{
    fprintf(m_fp, "**** Debugging complete of process "
            "%ld\n", (long)getpid());
//  fclose(m_fp);
}

void CTrace::Out(const char *fmt, va_list v)
{
    vfprintf(m_fp, fmt, v);
    fflush(m_fp);
}

void CTrace::Trace(const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);

    trace.Out(fmt, ap);
    va_end(ap);
}

void _dc_Trace(const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);

    vfprintf(stderr, fmt, ap);
    va_end(ap);
}
#endif
```

B.9.15 lib/src/mstring.C

```
/* MString: a string class */
#include "project.h"

// This is a library class that is only submitted because it is
// used by the rest of the code. It is very sparsely documented.

#include <string.h>
#include <ctype.h>
#include "mstring.h"

#define new DEBUG_NEW

const int STRING_GROW = 512;

MString::MString()
{
    m_pszString = NULL;
```

```
    m_nSpace = 0;
}

MString::MString(int nSize)
{
    m_pszString = new char[nSize + 1];
    *m_pszString = '\0';
    m_nSpace = nSize;
}

MString::MString(const char *pszString, int nCount)
{
    Fill(pszString, nCount);
}

MString::MString(const MString &str, int nCount)
{
    Fill(str, nCount);
}

MString::MString(char cChar)
{
    char szBuffer[2];
    szBuffer[0] = cChar;
    szBuffer[1] = '\0';
    Fill(szBuffer, 1);
}

MString:: MString()
{
    Empty();
}

void MString::MakeSpace(int nSize)
{
    if (nSize > m_nSpace)
    {
        char *pszNew = new char[nSize + STRING_GROW];

        if (m_pszString)
        {
            strcpy(pszNew, m_pszString);
            delete []m_pszString;
        }
        else
            pszNew[0] = '\0';
        m_pszString = pszNew;
        m_nSpace = nSize + STRING_GROW;
    }
}

void MString::Empty()
{
    if (m_pszString)
    {
        delete []m_pszString;
        m_pszString = NULL;
        m_nSpace = 0;
    }
}

void MString::Fill(const char *pszString, int nCount)
{
    if (pszString == NULL)
    {
        m_pszString = NULL;
```

```
        m_nSpace = 0;
    }
    else
    {
        if (nCount == -1)
        {
            m_pszString = new char[strlen(pszString) + STRING_GROW + 1];
            strcpy(m_pszString, pszString);
            m_nSpace = strlen(pszString) + STRING_GROW;
        }
        else
        {
            m_pszString = new char[nCount + STRING_GROW + 1];
            strncpy(m_pszString, pszString, nCount);
            m_pszString[nCount] = '\0';
            m_nSpace = nCount + STRING_GROW;
        }
    }
}

void MString::Fill(const MString &str, int nCount)
{
    if (str.m_pszString == NULL)
    {
        m_pszString = NULL;
        m_nSpace = 0;
    }
    else
    {
        if (nCount == -1)
        {
            m_pszString = new char[str.m_nSpace + 1];
            strcpy(m_pszString, str.m_pszString);
            m_nSpace = str.m_nSpace;
        }
        else
        {
            m_pszString = new char[nCount + STRING_GROW + 1];
            strncpy(m_pszString, str.m_pszString, nCount);
            m_pszString[nCount] = '\0';
            m_nSpace = nCount + STRING_GROW;
        }
    }
}

MString::operator const char *() const
{
    static char cNull = '\0';
    if (m_pszString == NULL)
        return &cNull;
    else
        return m_pszString;
}

MString &MString::operator=(const char *pszString)
{
    if (pszString)
    {
        int l = strlen(pszString);
        if ((l == 0) || (m_nSpace < l))
        {
            Empty();
            Fill(pszString);
        }
        else
            strcpy(m_pszString, pszString);
```

```
    }
    else
        Empty();
    return *this;
}

MString &MString::operator=(const MString &str)
{
    if(&str != this) {
        if ((str.GetLength() == 0) || (m_nSpace < str.GetLength()))
        {
            Empty();
            Fill(str);
        }
        else
            strcpy(m_pszString, str.m_pszString);
    }
    return *this;
}

MString &MString::operator+=(const MString &str)
{
    if (str.GetLength() > 0)
    {
        MakeSpace(GetLength() + str.GetLength());
        strcat(m_pszString, str.m_pszString);
    }
    return *this;
}

MString operator+(const MString &str1, const MString &str2)
{
    MString strTemp(str1.GetLength() + str2.GetLength());
    strTemp = str1;
    strTemp += str2;
    return strTemp;
}

int MString::GetLength() const
{
    return (m_pszString == NULL) ? 0 : strlen(m_pszString);
}

char *MString::Grab(int nSize)
{
    if (nSize != -1)
        MakeSpace(nSize);
    return m_pszString;
}

void MString::Release(int nSize)
{
    if (nSize != -1)
    {
        m_pszString[nSize] = '\0';
    }
}

bool operator==(const MString &str1, const MString &str2)
{
    return strcmp(str1.m_pszString, str2.m_pszString) == 0;
}

bool operator==(const MString &str1, const char *sz2)
{
    return strcmp(str1.m_pszString, sz2) == 0;
```

```
}

bool operator==(const char *sz1, const MString &str2)
{
    return strcmp(sz1, str2.m_pszString) == 0;
}

void MString::DropLeft(int nCount)
{
    char *p;
    char *q;
    char *pEnd;
    pEnd = m_pszString + strlen(m_pszString) - nCount;
    q = m_pszString + nCount;

    for(p = m_pszString; p <= pEnd;)
        *p++ = *q++;

}

void MString::DropRight(int nCount)
{
    char *p;
    p = m_pszString + strlen(m_pszString) - nCount;
    *p = '\0';
}

MString MString::Left(int nCount) const
{
    return MString(*this, nCount);
}

MString MString::Right(int nCount) const
{
    return MString(m_pszString + strlen(m_pszString) - nCount);
}

MString MString::After(int nStart) const
{
    return MString(m_pszString + nStart);
}

MString MString::Mid(int nStart, int nLength) const
{
    MString s(m_pszString + nStart);
    if (nLength < s.GetLength())
        s.m_pszString[nLength] = '\0';
    return s;
}

void MString::LTrim()
{
    const char *s = m_pszString;
    while (isspace(*s) && *s)
        s++;
    DropLeft(s - m_pszString);
}

void MString::RTrim()
{
    char *s = m_pszString + strlen(m_pszString);
    --s;
    while(s >= m_pszString && isspace(*s))
        *s-- = '\0';
}
```

```
void MString::Insert(int nStart, char ch)
{
    int l = strlen(m_pszString);
    MakeSpace(l + 2);

    if (nStart >= l)
    {
        m_pszString[l + 1] = '\0';
        m_pszString[l] = ch;
    }
    else
    {
        for(int i = l; i >= nStart; i--)
            m_pszString[i+1] = m_pszString[i];
        m_pszString[nStart] = ch;
    }
}

void MString::Remove(int nPos)
{
    char *p = m_pszString + nPos;
    while (*p)
        *(p++) = p[1];
}

int MString::Find(char ch, int nStart) const
{
    char *p;
    p = strchr(m_pszString + nStart, ch);
    return (p == NULL) ? -1 : p - m_pszString;
}

int MString::Find(const char *pszSearch, int nStart) const
{
    char *p;
    p = strstr(m_pszString + nStart, pszSearch);
    return (p == NULL) ? -1 : p - m_pszString;
}
```

B.9.16 lib/src/new.C

```
// Need to define new and delete here myself because otherwise electric fence
// doesn't work quite right with C++.

#include "project.h"

#undef new

class my_link
{
public:
    my_link *pPrev;
    my_link *pNext;
    char szFilename[32];
    int nLine;
    int nSeq;

    my_link() { pPrev = pNext = NULL; }
};

my_link Head;

static int nSeqCount = 0;

void *my_alloc(size_t s, const char *szFile, int nLine);
```

```
void my_free(void *);

// Should really throw an exception if malloc returns zero but g++ doesn't
// support exceptions too well.

void *operator new(size_t s)
{
    return my_alloc(s, "Unknown", -1);
}

void *operator new[](size_t s)
{
    return my_alloc(s, "Unknown array", -1);
}

void *operator new(size_t s, const char *szFile, int nLine)
{
    return my_alloc(s, szFile, nLine);
}

void *operator new[](size_t s, const char *szFile, int nLine)
{
    return my_alloc(s, szFile, nLine);
}

void operator delete(void *p)
{
    my_free(p);
}

void operator delete[](void *p)
{
    my_free(p);
}

void *my_alloc(size_t s, const char *szFile, int nLine)
{
    if (nLine == -1)
    {
        nLine = -2;
    }

    my_link *p = (my_link *)malloc(s + sizeof(my_link));
    strcpy(p->szFilename, szFile);
    p->nLine = nLine;
    p->nSeq = ++nSeqCount;

    p->pNext = Head.pNext;
    p->pPrev = &Head;
    Head.pNext = p;
    if (p->pNext)
        p->pNext->pPrev = p;

    return (void *)(p + 1);
}

void my_free(void *pv)
{
    if (pv)
    {
        my_link *p = (my_link *)pv;
        --p;

        p->pPrev->pNext = p->pNext;
        if (p->pNext)
            p->pNext->pPrev = p->pPrev;
```

```
        free(p);
    }
}

void dumpleaks()
{
    my_link *p = Head.pNext;
    if (p)
        TRACEX(("*LEAK* Memory leak dump...\n"));
    while (p)
    {
        TRACEX(("*LEAK* Object %d at %p, allocated %s:%d\n",
                p->nSeq, p+1, p->szFilename, p->nLine));

        if (!strcmp(p->szFilename, "mstring.C"))
        {
            const char *s = (const char *)(p + 1);
            TRACEX(("  String containing: '%s'\n", s));
        }
        p = p->pNext;
    }
}

class bing
{
public:
     bing() { dumpleaks(); }
};

static bing bingbing;
```

B.9.17 lib/src/sockutil.c

```
/*
 * Newwords vote system, socket utility functions
 *
 * Andrew Main <zefram@dcs.warwick.ac.uk>
 */

/* Very minor changes by Mike Crowe <mac@dcs.warwick.ac.uk> */

/* read(2) and write(2), but without the possibility of partial success *
 * if it is possible to continue.                                       */

#include <unistd.h>
#include <fcntl.h>
#include <signal.h>

#include "sockutil.h"

int xread(int fd, void *buf2, int n)
{
    register char *buf = buf2;
    int r, c=0;
    do {
        r = read(fd, buf+c, n-c);
        if(r == -1) return -1;
        c+=r;
    } while(r && c!=n);
    return c;
}

int xwrite(int fd, void *buf2, int n)
{
    register char *buf = buf2;
```

```
    int r, c=0;
    do {
        r = write(fd, buf+c, n-c);
        if(r == -1) return -1;
        c+=r;
    } while(r && c!=n);
    return c;
}

/* This puts a fd into blocking mode.  It's only needed on Solaris. */

void makeblock(int fd)
{
    int opt = fcntl(fd, F_GETFL, 0);
    opt &=  (O_NDELAY|O_NONBLOCK);
    fcntl(fd, F_SETFL, 0);
}

/* This sets SIGPIPE to be ignored.  Writes that would generate SIGPIPE *
 * will still fail with EPIPE.                                          */

void ignore_sigpipe(int ign)
{
    struct sigaction sa;
    sa.sa_handler = ign ? SIG_IGN : SIG_DFL;
    sa.sa_flags=0;
    sigaction(SIGPIPE, &sa, NULL);
}
```

Bibliography

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman (1986) *Compilers: Principles, Techniques and Tools*, Reading, Mass. : Addison-Wesley.

[2] Yosi Ben-Asher, Larry Rudolph and Dror G. Feitelson (1996) *ParC - An extension of C for shared memory parallel processing*, Software Practice and Experience Volume 26.5 Page 581.

[3] John Bloomer (1992) *Power Programming with RPC*, Sebastopol, CA : O'Reilly & Associates.

[4] Grady Booch (1994) *Object-Oriented Analysis and Design with Applications (2nd ed)*, Redwood, CA : Benjamin/Cummings.

[5] *USENET newsgroup: comp.compilers*

[6] N.H. Gehani and W.D. Roome (1989) *Concurrent C*, Summit, NJ : Silicon Press.

[7] G.A. Geist, J.A. Kohl, P.M. Papadopoulos (1996) *PVM and MPI: a Comparison of Features.*

[8] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Viady Sunderam (1994) *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*, Cambridge, Mass. : MIT Press.

[9] Sara Kalvala (1996) *Compiler Design lecture course*, University of Warwick.

[10] Xavier Leroy (1996) *Linuxthreads - POSIX 1003.1c kernel threads for Linux*, *ftp://ftp.xavier.fr/*.

[11] John R. Levine, Tony Mason and Doug Brown (1995) *Lex & Yacc*, Sebastopol, CA: O'Reilly & Associates.

[12] Cherri M. Pancake (1993) 'Multithreaded Languages for Scientific and Technical Computing', *IEEE Proceedings*, Volume 81 Page 288.

[13] Bruce Perens (1993) *Electric Fence Malloc Debugger*, *ftp.debian.org*.

[14] Dennis Ritchie (1993) 'The Development of the C language', *Second History of Programming Languages conference.*

[15] Bjarne Stroustrup (1991) *The C++ programming language*, Reading Mass. : Addison-Wesley.

[16] Bjarne Stroustrup (1994) *The Design and Evolution of C++*, Reading, Mass. : Addison-Wesley.

Index