Programming with GNU Software

Edition 2, 4 September 2002
$Id: gnuprog2.texi,v 1.22 2002/08/15 19:41:25 rmeeking Exp $

Table of Contents

Foreword

1 Introduction

The GNU Software Development Environment (GDE) is a full-featured and often coordinated development environment. It consists of a wide variety of largely independently developed tools that, as a whole, provide an environment that eases the tasks involved in producing high quality professional software. And, by the way, it happens to be freely down-loadable. Most of these tools have been developed by and for professionals and have been provided to the programming community for a wide variety of reasons. This book will generally ignore the reasons why people would develop these tools, and instead focus on why you would want to use them.

1.1 what this book is

This book is neither a comprehensive reference, nor a light weight novella. It is intended to be a learning guide for the GDE. The reader is presumed to be familiar with software development processes in general. That is, they are expected to be familiar with edit-compile-debug cycles on some sort of platform, whether that be UNIX or Windows or even something else. We will attempt to guide you through the process of choosing, configuring and using GNU tools. We will do this by developing an example project demonstrating the actual use of each tool.

There is an immense variety of tools available that can be classed as part of the GDE. Since this is not an encyclopedia of GNU tools, this book will be constrained to a small core set that serves as the foundation for GNU development: compilers, editors, build tools, source management tools and development libraries.

We will do this by introducing you to selected tools. The same ones the GNU developers use themselves. Each tool-based chapter will tell you how to obtain, configure, install and use the basic features of each of the tools it describes. This book is not intended as a long-term reference text for them, but it will get you through the basic introduction and fundamental usage of each one.

1.2 What the GDE is not

The GDE works on an amazingly wide variety of highly dissimilar platforms. This means it is likely to work on whatever platform you might have available, and it means it likely works on whatever platform you may wind up working on. This is a direct consequence of its history. The GDE tools began on a DEC Vax nearly 20 years ago. However, all the world is not a Vax; all the world is not a Sun; and Linux is a Johnny-come-lately. Many people in many places have been working out problems on many platforms for many years. The result is a robust collection of tools that could never have been developed by any one manufacturer and that provide a fairly uniform development environment for all these dissimilar platforms.

The net result is that we do not have an IDE (Integrated Development Environment). What we do have is a *Non*-Integrated Development Environment that works similarly across this variety of platforms yielding an integrated development experience. And, we have tools that release on their own schedules, which can sometimes cause problems. They do, generally, work together, but their separateness is both a strength and weakness. It is not always seamless. Interdependency is reduced, reducing overall complexity and schedule/release issues.

Not to say they don't play together, however. They build on each other and often do include cooperation hooks. Not always, though. Some Linux distributions supply a packaging layer that superimposes some interdependency checks. This vastly simplifies package upgrades.

Now that you know how widely usable the GDE is, this is a warning that we will not be dealing with portability issues. The tools do not mask over everything and the focus of this

book is the tools, not cross platform portability. Our primary example platform will be Linux, but very little will be said that is not directly relevant to using GDE on other platforms.

1.3 Audience

This book in intended for software developers who are unfamiliar with GDE tools and want to understand how to use them. They will find this book very useful. The reader is expected to know how to use a text editor, understand the C and C++ languages, but it should not be necessary to understand the UNIX build environment. To take full advantage of this book, however, it would be very helpful to have access to a UNIX or Linux machine with the GDE installed.

1.4 this book's organization

This book is not organized along the lines of "how to set up a development project." This book presumes the reader is moderately proficient in the skills of software development. Instead, the chapters are grouped by areas of relevance and ordered to the extent that if one chapter depends upon knowledge from another, it will generally follow the other chapter.

That said, this book does not need to be read sequentially. In fact, I don't recommend it. Each chapter will introduce you to one or a few tools that make up the core of the GDE and will not rely heavily on material from preceeding chapters. The first (this) chapter and the last chapter are the only departures from this scheme. The last chapter will be used to help guide you to and through the labyrinth of GDE software not covered here, but available on the net.

A reasonable approach to this book might be to skim the introductory paragraphs of each chapter, getting a clearer picture of what you can learn from each. Then, as you acquire, install or have need of specific tools, go back and read the chapter as you concurrently work with the tool. Instant feedback is a great way to reinforce learning and comprehension.

1.5 What You Are Presumed to Know

If you are familiar with unpacking a tarball[1] and configuring and building the result, then you can skip this section.

This book will deal with source code distributions. Some distributions are made as pre-packaged binary (pre-compiled) distributions, ready for installation on your system. We will not be covering those, as your distribution is likely to describe the installation in careful detail. Since the following chapters presume you have the knowledge, we describe the methods and requirements here as an introductory section.

First, to unpack and build a tarball, you must have the following development tools installed on your system.

'`tar`' This is an archiving and archive retrieval program. Basically, it stores a collection of file names, data and attributes in a single larger file. The data format is almost universally understood and comes as a standard utility on nearly all POSIX systems, Windows excepted.

'`cc`' All of the packages we deal with in this book are either interpreted programs (i.e., not compiled), or they require a C compiler. If your system does not come with a C compiler, you will have to obtain a pre- built one, even if it is non-ANSI. Once

[1] A *tarball* is a compressed '`tar`' archive, generally compressed with the '`gzip`' or '`bzip2`' utilities. '`gzip`' compression is far more common.

you have at least a rickety compiler, you will be able to build GCC, See Chapter 4 [The GNU Compiler Collection], page 75. Most compiled packages require an ANSI compiler, though GCC carefully does not.

'`make`' Most packages will build with a reasonably conventional MAKE program. If you do not have MAKE or if it is very old, you may have to download a pre-built binary for this program, too.

'`sh`' You cannot do much of anything without a Bourne-compatible shell program. All of the packaging and building requires such a shell program to process various command line commands in an automated fashion. ZSH is pretty close and often can work, but CSH and TCSH are sufficiently peculiar that they are very tricky to get working correctly, so they are not used in the scripts. You need to have available a tried and true Bourne derived shell, viz., SH, KSH, or BASH.

All of these tools can be obtained by going to this web site:

`http://www.gnu.org/software/software.html#HowToGetSoftware`

and following links to the source or binaries you need. However, if you have trouble obtaining or building these tools, there are several purveyors of GNU pre-built tools that will make your life much easier.

Once these tools are installed, then it becomes possible to build and install the various tools described in this book. *Using* the tools will also require PERL5, though, if you do not already have it. Since most programmers don't have strong need of Perl programming, it will not be covered in this book, but autoconf and automake require the Perl-5 interpreter for preparing your development project for building. **Note**: Perl is *not* required for actually building your product, unless you are using Perl sources yourself. Perl is only used to construct the make files.

Now, you have all your tools in place. Even Perl. To build and install any of the other tools described in this book, you need to perform the following steps:

- Acquire the '`mumble-1.2.3.tar.gz`' package for version 1.2.3 of the '`mumble`' tool and put it somewhere.
- Decide where you want to build and where to install the built product and do the following:
- '`cd /path/to/source`'
- '`gunzip -c /path/to/tarball/mumble-1.2.3.tar.gz | tar xvf -`'
  The '`tarball`' has now been unpacked into your source directory. You may build here if you wish, though it is often convenient to separate the source and build directories so that it is easy to distinguish between built files and source files.
- '`cd /path/to/build/dir`'
- '`prefix=/path/to/install/dir`'
- '`sh /path/to/source/mumble-1.2.3/configure --prefix=$prefix`'
  The product build instructions have now been customized for building on the current platform. (Many products can be cross built for alternate platforms. We won't be covering that here.)
- '`make`'
  If this step completes successfully, congratulations! You probably have a working product. "Probably" because there are so many platforms that it is possible yours was inadequately tested. This is actually highly unlikely for the widely used tools discussed in this book, but it is good practice to sanity check the product before installing it. Nearly all GNU-type products have a sanity check available:
- '`make check`'
  If this is successful, then we are now pretty sure it really does work the way it is supposed to.

- '`make install`'
  If you do not specify '`--prefix`' or specify a root-owned directory, you will likely need to perform this last step as the super user (root). Just make sure the '`$prefix/bin`' directory is in your '`PATH`' environment variable.

That's it. Now you can MUMBLE on your platform.

1.6 Conventions Used in This Book

The following conventions are used:

'`Italic`' Represents file and directory names, function names, program variables, names of books and of chapters in this book, and general emphasis. In examples, it is used to insert comments that are not part of the text you type in. *This is in italics.*

'`Bold`' Represents command names, options, keyboard keys, user names and C preprocessor directives, such as **#if**.

'`Constant Width`'
Represents programming language keywords such as `int` and `struct`. In examples, it is used to show program code, input or output files, and the output from commands and program runs.

'`Constant Width Bold`'
Used in examples to show commands or input that you enter at the terminal. e.g., "$ `ls`"

'`Constant Width Italic`'
Used in examples to show generic (variable) portions of a command that you should replace with specific words appropriate to your situation. For example:

```
rm filename
```

means to type the command **rm**, followed by *`filename`*, the name of a file.

'`$`' Used to show the shell prompt. The default shell prompt is different for different shells and can be changed by the user.

Two different notations are used in this book to represent the use of control keys. One is more familiar to most readers: the notation **CTRL-X** means you must hold down the **Control** key while typing the character "x". When discussing editor usage, we use the notation that its documentation uses: **C-x** means the same as **CTRL-X**. The shift key is irrelevant when you type a letter along with the **Control** key.

We denote other keys similarly (e.g., **Return** indicates a carriage return). All user input should be followed by a **Return**, unless otherwise indicated.

2 The GNU C Library

The beating heart of the entire GNU system is the GNU C language runtime: *glibc*. All of the various applications that comprise the GNU development environment (indeed the entire GNU system) call upon the services of the GNU C library one way or another. In this chapter we will talk about some of the GNU extensions to the usual C APIs that you have access to only when you use the GNU C library. We do not aim to be exhaustive, and we won't even cover all of the GNU extensions, but the reference manual that ships with GNU libc has details of every function and macro that *glibc* implements. Our aim here is to give you a flavour of the extra functionality you get when you use the GNU implementation of the C runtime library.

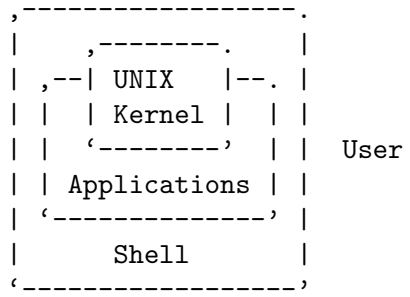
Throughout this book we talk about various extensions implemented by the GNU system, by which we mean additional features or programming interfaces you will have access to when you use a GNU system, beyond those available in a vanilla UNIX environment. More specifically, in this chapter we will be describing mostly *glibc extensions*, which are some of the extra features the GNU C library gives you access to.

Although *glibc* works best in conjunction with GCC, the library is carefully written to be used with almost any modern, standards conformant C compiler. Where we mention GCC extensions, we are referring to additional C language features you will have access to only when you use GCC; if, for some reason, you are using *glibc* but not GCC, you will of course be able to use *glibc* extensions in the context of the C language features provided by your own compiler, but not have the advantage of using any GCC extensions.

Historically, the fundamental design for the APIs that comprise the standard C library was set out in the late 1960's. Since then, not only has computer science come a long way, but the operating environment that the library codifies has changed considerably. The GNU C library authors have added features to address these changes to some extent, but also to make *glibc* somewhat more pleasant to use than it would be without the extensions they have added. If it is important that your code needs to compile and build in a non-GNU environment, then using the *glibc* extensions could give you a headache in the long run. This is certainly mitigated to a very large extent by the fact that *glibc* itself is extremely portable: installing *glibc* on the target machine is often an easier option than trying to write your code without the benefits gained from the *glibc* extensions.

2.1 Glibc Architecture Overview

Depicted in *example 2.1* is a diagram that is all too familiar to anyone who has studied CS101[1]. It stretches the metaphor of 'a nut' somewhat: the user interacts with the *shell*, which calls applications on the user's behalf, where the applications in turn call upon services tendered by the *kernel*.



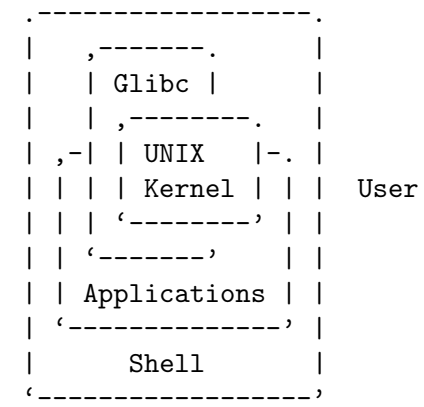
[1] Other text books depict the *kernel* entirely within *applications*. Strictly, by virtue of being an application itself, the *shell* does have access directly to the *kernel*, so we have shown that in our diagram.

Example 2.1: *Simplified system architecture component relationships*

Traditionally, the *shell* is represented by the command line interface that the user manipulates to control their machine – effectively your login shell, probably '`/bin/bash`', on a GNU/Linux system; or even DOS on a Windows machine. In reality graphical user shells are rife in modern computing, and much of the work of a traditionalist's *shell* is performed graphically. If you have GNU/Linux, then you almost certainly use either the GNOME or KDE desktops as your shell. Alternatively, if you use a Microsoft system, you might have noticed that DOS has been superceded by Windows as the dominant user shell in recent years...

At the centre of any 'nut' is, of course, a *kernel*. Again, on a GNU/Linux system, the metaphorical nut that is your computer system might have Linux as its kernel, or as another example: BSD UNIX if you run *Mac OS X*. Each of the many flavours of Windows also contain a kernel, but it is harder to draw a dividing line between shell and kernel in this case, since the Windows kernel provides both system services and graphical desktop management.

The *applications* are the flesh of our 'nut', and thus lie between the kernel, which controls the hardware in your computer, and the shell, which you interact with. Traditionally, an application accepts input text from a keyboard, and displays output text to a monitor screen. These days, applications tend to come with more sophistacted interfaces, and could accept input from a variety of devices: a mouse, a stylus or a scanner for instance. Equally, the output from an application likely involves not only displaying high resolution graphics on the monitor screen, but maybe also output to, among others, an LED display or a network interface card. The shell is itself really just another application that is specialised for helping a user to control and receive responses from other applications.



Example 2.2: *Simplified component relationships with glibc*

Here, in *example 2.2* we show where *glibc* fits in to all of this. Fundamentally, it provides a higher level API for interacting with the machine than the low level system calls implemented by the kernel. But more importantly than that, it supplies a standard interface to many of the common facilities used by the *applications*. It would be perfectly possible to write an application that does not use *glibc*, but implements everything from first principles in terms of the system calls provided by the *kernel*[2]. And yet, it would be a rare program that is designed in this way when the higher level interface of *glibc* requires fewer lines of source code to achieve the same ends.

[2] As a matter of fact, the kernel itself is one such application.

2.2 Standards Conformance

Over the years UNIX has undergone many revisions, and on occasion has split into independent developments from the common base line. One of the most fundamental of these historical code forks was between what we call BSD, the academic *Berkeley System Distribution* of UNIX, and the commercial *System V* distribution described by the *System V Interface Definition* (SVID). Each of these developments introduced new features to the C library that were not necessarily mirrored in the other: BSD developed APIs for sockets and signal handling for example; among others SVID introduced APIs for inter-process communication (IPCs), and shared memory management. Luckily, with a few exceptions, the new APIs written by these two developments are not mutually exclusive, and thus modern UNIX C libraries, including the GNU C library, support both.

The task of recombining the BSD and SVID flavours of UNIX into a single specification was done by the *International Standards Organisation*, when they ratified ISO/IEC 9899:1990 (commonly known as ISO C) from the earlier ANSI C standard of 1989. The GNU C library complies with this standard wherever it is applicable to the standard C library.

More recently, the *Portable Operating System Interface* (POSIX) was issued as ISO/IEC 9945-1:1996. It builds upon and is a superset of the ISO C standard, and includes a detailed specification of the requirements for a portable C library interface. Most developers who care about the portability of their code, write to this standard. The GNU C library also conforms to this standard.

Also of note is the GNU C library's conformance to the so called ISO C99 standard. This standard has not yet gained large scale acceptance, so although its features are available to you if you use GCC and the GNU C library, you may find that if you do use them, few other environments will be able to compile your code for the time being.

This book does not aim to teach the facilities that are described in these standards documents: See Section 5.8 [Further Reading], page 125, for details of some books we recommend if you need to learn about the standard C library APIs.

2.3 Memory Management

One of the most error-prone aspects of programming in C is the management of memory. As a programmer, the C language gives you very little help with tracking dynamic memory[3], and consequently you are forced to handle the low level details manually in your code.

The GNU system has a number of answers to solve a subset of the problems that normally require you to `malloc` and `free` blocks of memory at runtime, and hence reduce the overhead of managing memory for your application, thus reducing the complexity of your code.

2.3.1 alloca

Often, the only reason that you are forced to allocate memory from the heap[4] is because the amount of memory required cannot be calculated in advance (otherwise you could just use an array). Unfortunately, on many occasions, to prevent memory leaks, you have to remember to release that memory just before each exit point from the function that allocated it. The GNU system supports use of the function '`alloca`', which works almost identically to '`malloc`', except that the memory it returns is automatically released when the function exits. It even works with non-local exit functions such as '`longjmp`'.

[3] Memory allocated from the heap at runtime with the `malloc` family of functions.

[4] The *heap* is the pool of unused (virtual) memory that the operating system allocates to programs in pieces on request.

`void` ***alloca** (`size_t` *size*)* [glibc function]

Return the address of a block of *size* bytes of dynamically allocated memory.

Strictly speaking, the '`alloca.h`' header is shipped as part of *glibc*, but the '`alloca`' call is not a *glibc* extension – when '`alloca`' is encountered in your code, it is open coded by GCC. There is also slower version written in C ((**FIXME:** *alloca.c uri.*)) that can be linked with your application if you are not compiling with GCC. It is good practice to ship this file with the sources for your project so that your users will be able to compile your code even if their compilation environment doesn't support '`alloca`' natively.

In *example 2.3* there is a short function to test whether a named file exists in a particular directory. Notice how even though there is only one exit point from the function, using '`malloc`' to set aside some dynamic memory spoils the flow of the function. We have to save the return value of the call to '`access`' so that the memory can be manually released with '`free`'.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
...

int
file_exists (const char *dirpath, const char *filename)
{
  size_t len      = 1+ strlen (dirpath) + 1+ strlen (filename);
  char * filepath = (char *) malloc (len);
  int    result;

  if (filepath == 0)
    perror ("malloc");

  sprintf (filepath, "%s/%s", dirpath, filename);

  /* Invert the return status of access to behave like a boolean.  */
  result = 1+ access (filepath, X_OK);

  free (filepath);
  return result;
}
```

Example 2.3: *Checking whether a file exists – malloc version*

In *example 2.4* it is much easier to tighten up the code because we know the memory will be released automatically when the function has finished[5].

```
...
#include <alloca.h>
...

int
file_exists (const char *dirpath, const char *filename)
{
```

[5] '`alloca`' uses memory on the function call stack, so that when the stack frame for the function is removed at runtime, any additional memory set aside in that stack frame by '`alloca`' is automatically released.

```
  size_t len      = 1+ strlen (dirpath) + 1+ strlen (filename);
  char * filepath = (char *) alloca (len);

  sprintf (filepath, "%s/%s", dirpath, filename);
  return 1+ access (filepath, X_OK);
}
```

Example 2.4: *Checking whether a file exists – alloca version*

The drawback to using '`alloca`' is the lack of error reporting if there is insufficient memory to fulfill the request. In the event that the operating system runs out of stack space when trying to satisfy the '`alloca`' call, your application will simply crash – probably with a segmentation fault[6].

2.3.2 obstack

Another problem with the traditional '`malloc`' API is that it is very difficult to cope with chunks of memory that may expand during runtime, or are of unknown length when the call to '`malloc`' is made. If you need to keep the memory over a function call boundary, '`alloca`' is of no use.

Typically, this is a problem when reading strings into an application – either you have to scan the string once to calculate the length and then again to copy it into a correctly sized block of memory; or you have to call '`realloc`' to fetch a bigger block of memory each time you detect that you are about to go out of bounds. *example 2.5* uses the second of these methods.

```
char *
read_string (FILE *input)
{
  size_t i       = 0;
  size_t size    = 10;          /* Take a guess.  */
  char * string  = (char *) malloc (1+ size);
  int    c;

  while ((c = fgetc (input)) != EOF)
    {
      if (isspace (c))
        break;

      if (i == size)
        {
          size  *= 2;
          string = realloc (string, 1+ size);
        }

      string[i++] = (char) c;
    }
  string[i] = '\0';

  return string;
}
```

[6] GNU/Linux has as little as 2Mb of stack space in multi-threaded applications.

Example 2.5: *Reading a string into memory – malloc version*

In effect '`obstack`'s manage the resizing of allocated memory for you, albeit in a far more efficient manner than the manual '`realloc`'ation from *example 2.5*. All of the functions and macros described in this section can be accessed by including the '`obstack.h`' header in your file.

struct obstack [data type]

An opaque handle for an '`obstack`'. All of the functions for managing '`obstack`'s take a pointer to one of these structures. You can have as many '`obstacks`' in your program as you like, and each is capable of holding many strings. However there can be only one growing string in each '`obstack`' at any given time – when that string is complete, you can finalise it and start another string in the same '`obstack`'. As soon as you have done that, the finalised string cannot be changed. You are not limited to strings in fact, you can store any kind of growing memory object in an obstack, provided that only one object in each '`obstack`' is active. Hence the term *stack*: the active object is the top "plate" on the stack and can be accessed and changed, but you can't get to the plates underneath.

Before you can call any of the '`obstack`' functions, you need to decide how those functions will allocate and release dynamic memory, and must define the following macros in your source file:

```
#define obstack_chunk_alloc  malloc
#define obstack_chunk_free   free
```

You can define these macros to use any memory allocation scheme, though you must define them before any other calls to '`obstack`' functions. Despite our use of '`malloc`' and '`free`' to manage memory, '`obstack`'s request and release memory in large chunks which is more time efficient.

`int` **obstack_init** (`struct obstack*`*`obstack_handle`*`)` [glibc function]

This function initialises the '`obstack`' such that it is ready to have objects stored in it. This function must always be called to initialise an '`obstack`' before it can be used with any of the other functions detailed in the rest of this section.

```
#include <obstack.h>

#define obstack_chunk_alloc  malloc
#define obstack_chunk_free   free

static struct obstack *string_obs = NULL;
...

char *
read_string (FILE *input)
{
  if (string_obs == NULL)
    {
      string_obs = (struct obstack *) malloc (sizeof (struct obstack));
      obstack_init (string_obs);
    }

  while ((c = fgetc (input)) != EOF)
```

```
    {
      if (isspace (c))
        break;

      obstack_1grow (string_obs, (char) c);
    }

  return (char *) obstack_finish (string_obs);
}
```

Example 2.6: *Reading a string into memory – obstack version*

In *example 2.6* we keep adding characters to the growing object as soon as they are read in from the '`input`' stream. The '`obstack`' takes care of ensuring that there is enough memory to add the characters, internally fetching more memory from the system using `obstack_chunk_alloc` as necessary. Consequently the object might move occasionally as it is growing, so it is important not to rely on the address of the object before it has *finished*.

Remember that only one object in the '`obstack`' can be growing at any given time. The object is started implicitly as soon as any bytes are added to it with the following function calls, but must be finished explicitly using '`obstack_finish`'.

`int` **obstack_1grow** (`struct obstack*`*`obstack_handle`*`, char` *`ch`*) [glibc function]
Simply grow the current object in *obstack_handle* by a single byte, *ch*.

`void` ***obstack_finish** (`struct obstack*`*`obstack_handle`*) [glibc function]
Declare that the current object has finished growing, and return the address of the start of that object. The next time one of the grow functions is called for *obstack_handle*, a new object will be started.

There are also other functions for growing the current object in an '`obstack`' by a different size than a single byte:

`int` **obstack_ptr_grow** (`struct obstack*`*`obstack_handle`*`, void *`*`data`*) [glibc function]
Grow the current object in *obstack_handle* by sufficient size to hold a pointer, and fill that space with a copy of *data*.

`int` **obstack_int_grow** (`struct obstack*`*`obstack_handle`*`, int` *`data`*) [glibc function]
Grow the current object in *obstack_handle* by sufficient size to hold an `int`, and fill that space with a copy of *data*.

`int` **obstack_grow** (`struct obstack*`*`obstack_handle`*`, void *`*`data`*`, int` *`size`*) [glibc function]
Grow the current object in '`obstack_handle`' by *size* bytes, and fill that space by copying *size* bytes from *data*.

`int` **obstack_grow0** (`struct obstack*`*`obstack_handle`*`, void *`*`data`*`, int` *`size`*) [glibc function]
Grow the current object in '`obstack_handle`' by *size* bytes, and fill that space by copying *size* bytes from *data*, followed by a null character.

`int` **obstack_blank** (`struct obstack*`*obstack_handle*`, int` *size*) [glibc function]
Grow the current object in '`obstack_handle`' by *size* bytes, but leave them uninitialized. You can also shrink the current object by passing a negative value in *size*, if you are careful not to reduce the size of the object below zero.

`int` **obstack_object_size** (`struct obstack*`*obstack_handle*) [glibc function]
Return the size of the growing object. This function is useful for ensuring that you don't decrease the size of an object below zero with '`obstack_blank`'. If the current object has not yet been grown (for example immediately after a call to '`obstack_finish`'), this function will return zero.

2.3.3 argz

2.4 Input and Output

2.4.1 Signals

UNIX systems send signals to programs to indicate program faults, user-requested interrupts, and other situations. More generally, signals are a simple interprocess communication – an aspect that's taken much further with the extended signal facilities that POSIX.4 defines. For the most part, signals are simply the way in which the operating system informs the process of an action that requires attention – whether it's a memory access violation or an external event. However, processes can also signal each other when that is useful.

Many signals are purely informative; others cause the program to change its state in some way. For example, when you enter *`Ctrl C`* at the keyboard, your shell will send the signal `SIGINT` to the foreground process, which asks the program to terminate. Or, if the program attempts an illegal memory access, the operating system sends it the signal `SIGSEGV`.

When a signal arrives, the program takes one of the following actions:

Ignore It ignores the signal and keeps running as if nothing happened.

Terminate It terminates, possibly leaving a core dump.

Stop It stops running, in a way that allows the program to be restarted.

Continue If the program is currently *stopped*, then it resumes running.

Execute handler
It executes some signal-handling routine previously installed by the program. (This action is never the default.)

Each signal has a default action, which determines what effect the signal has when it arrives, but a program can install a non-standard action for most signals which then overrides the default action. The signals on any system are defined in the header file '`signal.h`'. They vary from system to system, though most UNIX systems have some superset of 32 standard signals. The following important signals are almost always available:

Signal	Default Action	Meaning
`SIGHUP`	Terminate	Hangup; sent when the system asks a program to cleanly exit
`SIGINT`	Terminate	Interrupt; often sent when user types *`Ctrl C`*
`SIGQUIT`	Terminate	Quit; often sent when a user types *`Ctrl \`*
`SIGILL`	Terminate	Illegal instruction

SIGTRAP	Terminate	Sent when a program reaches a previously set breakpoint
SIGABRT	Terminate	Sent when the program calls abort
SIGFPE	Terminate	Floating point arithmetic error
SIGKILL	Terminate	Enforced program termination
SIGUSR1	Terminate	User defined signal 1
SIGUSR2	Terminate	User defined signal 2
SIGSEGV	Terminate	Segmentation violation; illegal memory access
SIGPIPE	Terminate	Broken command pipe
SIGALRM	Terminate	Alarm clock signal
SIGTERM	Terminate	Termination request (sent by software)
SIGCHLD	Terminate	Child process has stopped or terminated
SIGCONT	Continue	Continue processing if stopped
SIGSTOP	Stop	Stop processing; sent by the system
SIGTSTP	Stop	Stop processing; often sent when a user types *Ctrl Z*
SIGTTIN	Stop	Background program requires input
SIGTTOU	Stop	Background program cannot output

Of the signals listed above, SIGKILL and SIGSTOP are special: they are *unblockable*. That is, a program cannot change the handling of these signals to be different from the default. When a program receives a SIGKILL, it is always stopped dead in its tracks; similarly a SIGSTOP always causes the program to stop processing (and wait for a SIGCONT to wake it up again.

It's also worth noting that SIGUSR1 and SIGUSR2 are provided specifically as user-definable signals for your applications. These signals have no predefined default meaning, and are never sent to a process by the operating system.

There are functions for dealing with signals: raise, which a program can use to send a signal to itself, kill for sending a signal to an arbitrary process, and sigaction[7] that a program uses to change the action for any signal.

raise is very simple; it has a single argument, which is the name of a signal (i.e., one of the constants defined above). Here's how it is used:

```
#include <signal.h>

int ret;
...
ret = raise (SIGINT);
```

Example 2.7: *Using the* raise *library call.*

This function call sends the signal SIGINT. The return value, ret, is 0 if raise is successful; -1 if it fails. However, note that a signal may terminate the process – in which case, raise will never return.

If you want to send a signal to an arbitrary process (rather than sending a signal to yourself), you need to call kill, which has two arguments: a process ID and a signal name. It works in much the same way as the *kill* command that you can run from the command line. The excerpt in *example 2.8* behaves in exactly the same way as *example 2.7* – the program sends a SIGINT signal to itself.

```
#include <signal.h>
#include <sys/types.h> /* For pid_t definition. */
```

[7] This function replaces signal, an older library call which is best avoided altogether since its behaviour varies between systems.

```
int ret;
...
ret = kill (getpid (), SIGINT);
```

Example 2.8: *Using the* `kill` *system call.*

As with `raise`, when `kill` returns, `ret` will be 0 if the call was successful, or -1 if it failed. Of course, with this example, if there is no handler installed for `SIGINT`, then the default action is to terminate the process, so `kill` won't actually return at all.

The function `sigaction` changes the action that a process takes when it receives a signal. It uses a structure to pass information about the handling of a signal:

```
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask;
  int sa_flags;
}
```

Under most conditions, the `sa_handler` field is used to describe the action to be taken on receipt of a signal. Finer control can be gained by setting the `SA_SIGINFO` bit of `sa_flags` and then using the `sa_sigaction` field instead of `sa_handler` to describe what action to take when the signal arrives. We will describe how to use `sa_handler` style signal handling here: Check your system manual pages, or the *glibc* manual for details of how to use `sa_sigaction`.

You must set `sa_handler` to one of the following values:

'`SIG_DFL`' Restore the handling of this signal to the default (as shown in the earlier table).

'`SIG_IGN`' Ignore any instances of this signal.

'`handler`' When this signal arrives, execute the named handler function, which must have a prototype like this:

> `void` **handler_function** (`int` *signum*) [signal handler]
> The number of the signal that triggered the call to `handler_function` will be passed as *signum*.
>
> If the handler exits by calling '`return`', the program continues executing after receiving the signal. If the handler calls '`exit`' or '`abort`', execution doesn't continue after receiving the signal.

While executing the action for a signal, the arrival of subsequent signals with the same number is blocked – unless the `SA_NOMASK` bit of *sa_flags* is set to prevent that behaviour. Additional different signal numbers can also be blocked during the execution of the action by adding them to the *sa_mask* field of the `struct sigaction`. If you find you need to do this, your system manual page for `sigprocmask` explains how to manipulate `sigset_t` types.

Here's a very simple program that exercises the signal handler:

```
#include <signal.h>
#include <string.h>

void
sighandler (int signal)
{
```

```
  printf ("received signal %d\n", signal);
  abort  ();
}

int
main (int argc, const char *argv[])
{
  struct sigaction action;

  memset (&action, 0, sizeof (struct sigaction));
  action.sa_handler = sighandler;

  if (sigaction (SIGINT, &action, NULL) != 0)
    perror ("sigaction");
  sleep  (60);
}
```

Example 2.9: *Setting a simple* `signal` *handler.*

The program installs the function `sighandler` as an interrupt handler for '`SIGINT`'; then it sleeps, waiting for the user to type *CTRL-C*. When you type *CTRL-C*, `sighandler` is called; it prints its message, then calls `abort` to terminate execution. Here's how it looks:

```
$ ./a.out
CTRL-Creceived signal 2
Abort (core dumped)
$
```

You can install a separate signal handler for each signal that you care about; or else, since the argument passed to the handler identifies the signal that triggered this call, you can write a single handler function and let it figure out what to do on the basis of that argument.

2.4.2 Time Formats

Knowing the current time is important in many programs for various reasons. On UNIX systems, the '`time`' call gets the current time in seconds from the operating system. The time is simply a long integer that contains the number of seconds since an arbitrary moment – midnight at the beginning of January 1, 1970[8] – referred to as the *epoch*.

`time_t` **time** (`time_t *`*clock*) [system function]
This function returns the number of seconds elapsed since the epoch, and additionally stores the same number in the memory pointed to by *clock* (unless *clock* is `NULL`).

`char` ***ctime** (`time_t *`*clock*) [glibc function]
Returns a canonical 26-character string that displays the time represented by the integer in *clock* – for example '`Tue Mar 19 19:39:46 2002`'.

The '`time`' and '`ctime`' calls are precise enough for mundane tasks like displaying the date and time to the user, but you have to convert the `time_t` into the more meaningful '`struct tm`' format in order to measure elapsed time accurately:

[8] In the unlikely event that we are still using computers that are limited to 32-bits by then, this value will eventually wrap around one second after 3:14:07am January 19th, 2038.

```
struct tm {
  int tm_sec;     /* measured from 0 to 60 (incase of leap second) */
  int tm_min;     /* measured from 0 to 59 */
  int tm_hour;    /* measured from 0 to 23 */
  int tm_mday;    /* day of the month, measured from 1 to 31 */
  int tm_mon;     /* month, measured from 0 to 12 */
  int tm_year;    /* year, where zero is 1900 */
  int tm_wday;    /* day of the week, measured from 0 to 6 */
  int tm_yday;    /* day of the year, measured from 0 to 365 */
  int tm_isdst;   /* 1 is daylight saving time, else 0 */
  char* tm_zone;    /* name of timezone */
  long tm_gmtoff; /* timezone's distance from UTC in seconds */
};
```

Each element of the time that you'd be interested in is contained in a separate member of the structure. Watch those integers! They're not consistent. The day of the month is measured starting at 1, whereas the other integer values start at zero.

The calls listed below are used to convert from `time_t` to `struct tm` types, and then turn them into human readable strings. They all require you to include the file '`time.h`'.

`struct tm` ***gmtime** (`time_t *`*clock*`*`) [glibc function]
Breaks down the time reported by *clock* into a '`tm`' structure, using the UTC (Greenwich Mean time) time zone. The returned structure points to static memory which is overwritten by subsequent calls, making this function unsafe in a multi-threaded program.

`struct tm` ***gmtime_r** (`time_t *`*clock*`, struct tm *`*brokentime*`*`) [glibc function]
Much the same as '`gmtime`', except that the broken down time is stored in the memory pointed to by *brokentime*, to avoid problems with multi-threaded programs. This function conforms to the POSIX standard, and is thus supported by most modern platforms.

`struct tm` ***localtime** (`time_t *`*clock*`*`) [glibc function]
Much the same as '`gmtime`', except the conversion is performed for the local time zone.

`struct tm` ***localtime_r** (`time_t *`*clock*`, struct tm *`*brokentime*`*`) [glibc function]
Again, in accordance with POSIX, as '`localtime`', except that the broken down time is stored in the memory pointed to by *brokentime*.

Universal Coordinated Time or UTC [*sic*] used to be known as Greenwich Mean Time. It's the worldwide standard for reporting the time, using the local time in Great Britain and not recognizing daylight saving time. If you want to get UTC (we don't know why you would) you can call '`gmtime`' or '`gmtime_r`'. For local time, use '`localtime`' or '`localtime_r`'.

Here is some simple code that retrieves the time by calling '`time`' and passing its return value to '`localtime_r`'. Then the code extracts the hour and minute and displays the time.

```
#include <time.h>
#include <stdio.h>
#include <strings.h>

int
main (int argc, const char *argv[])
{
  struct tm brokentime;
```

```
  time_t    clock;
  char      buffer[6];
  char      time_tmp[3];

  /* Fetch the current time. */
  time (&clock);

  /* Convert it to a broken down time. */
  localtime_r (&clock, &brokentime);

  /* Start filling buffer with the hour. */
  sprintf (buffer, "%2d", current_struc->tm_hour);

  /* Add a colon. */
  strcat  (buffer, ":");

  /* Finish with the minute, always making it two digits. */
  sprintf (time_tmp, "%2.2d", current_struc->tm_min);
  strcat  (buffer, time_tmp);

  printf  ("Time is (or was, 6 calls back) %s\n", buffer);

  return 0;
}
```

Example 2.10: *Simple use of the time call.*

In the *example 2.10* we've done this the long way, grinding out a format manually, so you'll appreciate being able to use '`strftime`':

`int` **strftime** (`char *`*buf*`, int` *bufsize*`, char *`*fmt*`, struct tm *`*brokentime*`)` [glibc function]

Converts the time contained in the *brokentime* structure to a string of no more than *bufsize* characters and stores it in *buf*, using the format specified by *fmt*.

`char *`**strptime** (`char *`*buf*`, char *`*fmt*`, struct tm *`*brokentime*`)` [glibc function]

The converse of '`strftime`'. Converts the time contained in *buf* to the *brokentime* structure, parsing *buf* in the format specified by *fmt*.

Now let's use '`strftime`' to create a string suitable for display. The first argument is a buffer to put the string in, and the second argument is the size of this buffer. You have to choose a format – here we've chosen hh:mm – and specify it in the third argument. The fourth argument is the '`tm`' structure.

```
(void) strftime (buffer, sizeof (buffer), "%H:%M", brokentime);

printf ("Time is %s\n", buffer);
```

Example 2.11: *Simple use of the strftime call.*

In the format we chose, %H stands for the hour and %M for the minute. We put a colon between them, which comes out literally in the display. You can format your string any way you want, putting in spaces, commas, and special characters like \n for newline.

Now that you see the concept behind the format, you can look at all the available specifiers in '`strftime`' and '`strptime`'. If you don't want to mess with individual specifiers, just use %c and you'll get the date and time in a reasonable format. If you check the %R entry, you'll see that we didn't even have to do all the work that was in the previous example.

'`%a`' Abbreviated day of the week in local format (for example *Tue*).

'`%A`' Full day of the week in local format (*Tuesday*).

'`%b`'
'`%h`' Abbreviated month name in local format (*Mar*).

'`%B`' Full month name in local format (*March*).

'`%c`' Date and time in local format (*Tue 19 Mar 2002 01:02:03 GMT*).

'`%C`' The century number, either 19 or 20 considering the range of dates covered by the 32-bit epoch offset in seconds.

'`%d`' Day of the month as an integer that can contain a leading zero to make up two digits, measured from 01 to 31.

'`%D`' Date in the format '`%m/%d/%y`' (*03/03/02*)[9].

'`%e`' Day of the month as an integer that can contain a leading blank if it is only one digit, measured from 1 to 31.

'`%F`' Date in the format '`%Y-%m-%d`'[10].

'`%H`' Hour as a two-digit integer on a 24-hour clock, measured from 00 to 23.

'`%I`' Hour as a two-digit integer on a 12-hour clock, measured from 00 to 11.

'`%j`' Day of the year as a three-digit integer, measured from 001 to 366.

'`%m`' Month as a two-digit integer, measured from 01 to 12.

'`%M`' Minute as a two-digit integer, measured from 00 to 59.

'`%n`' Newline, equivalent to '`\n`'.

'`%p`' '`AM`' (morning) or '`PM`' (afternoon).

'`%P`' '`am`' (morning) or '`pm`' (afternoon).

'`%r`' Time in the preferred local 12-hour clock format '`%I:%M:%S %p`'.

'`%R`' Time in the 24-hour clock format '`%H:%M`'.

'`%S`' Second as a two-digit integer, measured from 00 to 59.

'`%t`' Tab, equivalent to '`\t`'.

'`%T`' Time in the 24-hour clock format '`%H:%M:%S`'.

'`%u`' The day of the week as an integer, measured from 1 to 7, where Monday is the first day of the week.

'`%U`' Week of the year as a two-digit integer, measured from 00 to 53, where the first Sunday of the year is the first day of week '`01`', and any days in the preceding partial week are in week '`00`'.

[9] Outside of America '`%d/%m/%y`' is more common, so this specifier is ambiguous at best. '`%F`' is the preferred format.

[10] As specified by the ISO 8601 standard.

'`%V`' Week of the year, measured from 01 to 53, where Monday is the first day of the week, and week 1 is the first week that has at least 4 days in the current year.

'`%w`' Day of the week, measured from 0 to 6.

'`%W`' Week of the year as a two-digit integer, measured from 01 to 53, where the first Monday of the year is the first day of week '`01`', and any days in the preceding partial week are in week '`00`'.

'`%x`' Date in the preferred local format (Here in Britain that would be '`%d/%m/%y`').

'`%X`' Time in the preferred local format (Here in Britain that would be '`%h:%m:%s`').

'`%y`' Year of the century as a two-digit integer, measured from 00 to 99.

'`%Y`' Year, as a four-digit integer.

'`%z`' The time zone as an offset from UTC in hours[11].

'`%Z`' Time zone abbreviation (*BST*).

'`%%`' Produces a percent sign (%) in the output.

Example 2.12: *strftime format specifiers*

In addition to the standard set of format specifiers in *example 2.12*, the GNU C library provides a small number of additional specifiers:

'`%k`' Hour as an integer that can start with a blank, on a 24-hour clock measured from 00 to 23.

'`%l`' Hour as an integer that can start with a blank, on a 12-hour clock measured from 00 to 11.

'`%s`' Number of seconds since the epoch.

Example 2.13: GNU *C library extensions to strftime format specifiers*

The GNU C library

An integer value should be useful for calculating elapsed time, but most timing you'd want to do (in order to schedule tasks in your program, for instance) requires better accuracy than to the nearest second. The GNU C library also provides functions to manipulate '`struct timeval`' times with finer granularity than one second:

```
struct timeval {
  time_t tv_sec;  /* Seconds */
  time_t tv_usec; /* Microseconds */
};
```

The '`timeval`' structure is defined in the '`sys/time.h`' header file.

int gettimeofday (`struct timeval *tv, struct timezone *tz`) [glibc function]

This function is similar to the '`time`' function we described earlier, except that it fills in a '`struct timeval`' with the time elapsed since the epoch to the nearest microsecond, rather than setting a simple '`time_t`'. The *tz* parameter is obsoleted by the '`tm_zone`' member of the broken down time structure, but is retained for backwards compatibility: You should always pass a '`NULL`' for this argument.

[11] RFC 822 timestamps are '`%a, %d %b %Y %T %z`'.

Here is a simple program that demonstrates both the use of microsecond accuracy, and timezone information:

```
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

int
main (int argc, const char *argv[])
{
  struct timeval tv;
  struct tm      brokentime;
  char           buffer[80];

  /* Fetch the current time to the nearest microsecond. */
  gettimeofday (&tv, NULL);

  /* Convert the whole seconds component to broken down time. */
  localtime_r (&tv.tv_sec, &brokentime);

  /* Generate the majority of the time display format. */
  strftime (buffer, sizeof(buffer), "%a, %d %b %Y %T", &brokentime);

  /* Display with microseconds, and timezone. */
  printf ("%s.%d %s\n", buffer, tv.tv_usec, brokentime.tm_zone);

  return 0;
}
```

Example 2.14: *Demonstrating use of timezones and microsecond accuracy.*

When you compile and run this program, you will see something akin to the following:

```
$ gcc -o usec usec.c
$ ./usec
Tue, 19 Mar 2002 19:36:40.474770 GMT
$
```

2.4.3 Formatted Printing

Formatted printing from

2.5 Error Handling

UNIX programs always return an exit code. Exit codes usually go unnoticed, but are always present. They are most often used by shell scripts and by make, which may take some alternative action if the program doesn't finish correctly.

By convention, most programs return an exit code of zero to indicate normal completion. Nonzero exit codes usually mean that an error has occurred. There are some notable violations of this convention (e.g., cmp and diff), but we recommend you obey it. C shell users can print exit codes by entering the command:

```
% echo $status
```

If you use the Bourne shell (sh) or a derivative like bash, you can print the exit code with the command:

```
$ echo $?
```

When you are writing a program, to have it return an error code, call the function exit, with the error code as an argument:

```
int code;

exit (code);
```

Although the error code is an int, its value should be between 0 and 255.

Functions aren't that different from programs: their returned value indicates whether they succeeded or failed. If you want to write healthy code, you should always check the value returned from a function. The code below shows one way to test for an error:

```
#include <stdio.h>

/* write to standard output and check for error */
if (printf (...) == EOF) exit (1); /* output failed; terminate */
/* normal processing continues */
...
```

The manual page for printf says that it returns the constant '`EOF`' if it fails; if it succeeds, it returns the number of bytes transmitted. If you look in '`stdio.h`', you'll see that '`EOF`' is defined as -1. As a rule, a positive or zero return value indicates success; a negative value indicates failure. However, there are many exceptions to this convention. Check the manual page, and if a symbolic constant (like '`EOF`') is available, use that rather than a hard-wired constant.

If you want more information about what went wrong with your function, you can inspect the errno variable. When a system call fails, it stores an error code in errno. UNIX then terminates the system call, returning a value that indicates something has gone wrong – as we just discussed. Your program can access the error code in '`errno`', figure out what went wrong, and (possibly) recover, as in the example below:

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

FILE *fd;
int fopen_errno;

main (int argc, char **argv)
{
  /* try opening the file passed by the user as an argument */
  if ((fd=fopen (argv[1], "r")) == NULL)
    {
      /* test errno, and see if we can recover */
      fopen_errno = errno; /* save errno */
      if (fopen_errno == ENOENT) /*no such file or directory*/
        {
          /* get a filename from the user and try opening again */
          ...
}
```

```
      /* otherwise, it's an unknown error */
      else
        {
    perror ("Invalid input file");
    exit (1);
        }
    }
  /* normal processing continues */
}
```

Example 2.15: *Use of errno*

On UNIX systems, the header file '`errno.h`' defines all possible system error values. Note that this code immediately copies errno to a local variable. This is a good idea, since '`errno`' is reset whenever a system call from your process fails, and there's no way to recover its previous value.

Another way to use the error number, without examining it yourself, is to call '`perror`', which sends a message describing the error to the standard error I/O stream. It's used like this:

```
#include <stdio.h>

perror ("prefix to message");
```

The '`prefix to message`' given as an argument to '`perror`' is printed first; then a colon; then the standardized message. For example, if the previous function call failed with the errno '`ENAMETOOLONG`' because the user passed a long string of garbage as an argument, the function call '`perror ("Invalid input file");`' would result in the output:

```
Invalid input file: File name too long
```

`void` **error** (`int` *`status`*`, int` *`errnum`*`, const char *`*`format`*`, ...`) [glibc function]

`void` **error_at_line** (`int` *`status`*`, int` *`errnum`*`, const char *`*`filename`*`,` [glibc function]
`unsigned int` *`lineno`*`, const char *`*`format`*`, ...`)

2.6 Pattern Matching

In this context, a *pattern* is a way of expressing a set of strings, and *pattern matching* is the act of determining whether a particular string is in the set described by the pattern. For example, some matches for the set of strings containing digits would be "7", "123" and "November 4th". But not "November" or "fourth".

The GNU C library supports a number of pattern matching concepts, from very basic *wildcard* matching, through to *extended regular expression matching*. In this section we will describe each, and provide annotated examples.

Pattern matching is a concept that is central to the operation of modern operating systems. The GNU system provides several tools that use pattern matching: `grep`, `sed` and even the shell itself. The GNU C library provides the APIs for employing pattern matching techniques in your own programs.

2.6.1 Wildcard Matching

This is the kind of pattern matching will be immediately familiar if you have ever used a UNIX command shell. Here is a short snippet of shell code you might use to add a new directory to your `PATH`, unless that directory was already included:

```
case :$PATH: in
  *:/opt/gnu/bin:*) ;;
  *)                PATH=/opt/gnu/bin:$PATH ;;
esac
```

The Bourne shell '`case`' construct tries to match the argument string, expanded from '`:$PATH:`', against each of the patterns that follow, and executes the code in the first matching branch. Specifically, if the expansion of '`:$PATH:`' already contains the substring '`:/opt/gnu/bin:`', then the first empty branch is executed. Otherwise, the second branch is executed, and '`/opt/gnu/bin:`' is prepended to the existing value of '`PATH`'.

There are, then, two *patterns* in this '`case`' statement, the first describes the set of all strings that contain the substring '`:/opt/gnu/bin:`', and the second describes an infinite set that will match any string at all! Except for a few special *wildcard* characters, each character in the pattern must match itself in the test string in order for the pattern as a whole to match successfully.

As evidenced by the second pattern in the example above, the '`*`' wildcard character will match successfully against any string of characters, or indeed against no characters at all. And hence the other pattern in the example could be read as: "The set of strings which begin with zero or more characters followed by the substring '`:/opt/gnu/bin:`', followed by zero or more characters".

The idiom of adding an extra delimiter at either end of a test expression, for example the extra '`:`' in '`:$PATH:`' saves you from having to worry about needing to match the first and last items in the expansion of '`$PATH`' explicity. Without the extra '`:`'s, you would need to write this:

```
case $PATH in
  /opt/gnu/bin:*)   ;;
  *:/opt/gnu/bin:*) ;;
  *:/opt/gnu/bin)   ;;
  *)                PATH=/opt/gnu/bin:$PATH ;;
esac
```

In addition, wildcard patterns have a number of other special characters:

'`?`' This wildcard will successfully match any single character.

'`[`*ab*`]`'

'`[`*x*`-`*y*`]`' Square brackets describe *character sets* which will successfully match against any one of the characters listed explicitly, or within a range bounded by '*x*`-`*y*'. For example '`[0-9]`' will match against any single digit.

To match the character '`]`' with a character set, it must be the first character listed. Similarly, to match a literal '`-`', the '`-`' must be the first or last character listed in the set.

'`[:`*class*`:]`'

As an alternative to listing *character sets* explicitly, wildcard patterns can also specify some commonly used sets by name. Valid names are '`alnum`', '`alpha`', '`ascii`', '`blank`', '`cntrl`', '`digit`', '`graph`', '`lower`', '`print`', '`punct`', '`space`', '`upper`' and '`xdigit`', which correspond to the C functions '`isalpha`', '`isascii`' etc.

Each of these is an example of an *atom*, as are each of the non-wildcard self-matching characters. In the context of pattern matching, an atom is any minimal constituent part of the pattern that potentially matches one or more characters. For example '`a`' is an atom that matches the character '`a`'; '`[0-9]`' is an atom that matches any of the characters '`0`' through '`9`' (so is '`[[:digit:]]`'); and so on. None can be broken down into smaller valid sub-patterns.

Sometimes the set of strings you are trying to describe with a pattern should include a literal '`*`', or another of the characters that are interpreted as wildcards when you use them in a

pattern. By prefixing those characters with a '\' character, the special meaning is turned off (or *escaped*) allowing the character to be matched exactly. Hence, '\' is not an atom, since it cannot match anything by itself. Rather, '\' might modify the meaning of the character that comes right after it: '`\*`' is an atom that matches only the character '`*`'; '`\a`' will match only the character '`a`'; and '`\\`' matches the character '\'.

(**FIXME:** *ISTR some pecularity with " and character sets.*)

The GNU C library provides a function which will tell you whether specific strings match against a given pattern:

`int` **fnmatch** (`const char *`*`pattern`*`, const char *`*`string`*`, int `*`flags`*`)` [Function]
This function returns zero if *pattern* describes a set that contains *string*. Conversely, if *pattern* does not match *string*, this function returns non-zero.

The *flags* argument is a bit field built by bitwise *or*ing of the various options that tweak the behaviour of the matching process. Some of the values you could pass are as follows:

'`FNM_NO_ESCAPE`'
Normally, if you want to match a '\' character, you must write '\\' in *pattern* because the normal behaviour of '\' is to escape any special behaviour of the following character. Passing this flag makes '\' behave like a normal character, but leaves you with no way to escape special characters in *pattern*.

'`FNM_CASEFOLD`'
Passing this flag causes the pattern to make no distinction between upper and lower case characters when deciding whether *pattern* matches *string*.

'`FNM_EXTMATCH`'
When you pass this flag, you can use the following additional atoms in *pattern* to help describe the set of matching *string*s, where *pattern-list* is a list of '`|`' delimited patterns:

'`!(`*`pattern-list`*`)`'
This pattern matches if none of the patterns in *pattern-list* could match *string*.

'`*(`*`pattern-list`*`)`'
This pattern matches if zero or more occurences of the patterns in *pattern-list* match *string*.

'`?(`*`pattern-list`*`)`'
This pattern matches if zero or one occurences of the patterns in *pattern-list* match *string*.

'`@(`*`pattern-list`*`)`'
This pattern matches if exactly one occurence of any pattern in *pattern-list* match *string*.

'`+(`*`pattern-list`*`)`'
This pattern matches if one or more occurences of the patterns in *pattern-list* match *string*.

There are other flags that can be added to the *flags* argument of '`fnmatch`', which you can find in your system documentation.

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

```
int
main (int argc, const char *argv[])
{
  const char *pattern;
  int count;
  int arg;

  if (argc < 3)
    {
      fprintf (stderr, "USAGE: %s <pattern> <string> ...\n", argv[0]);
      return 1;
    }

  pattern = argv[1];

  for (count = 0, arg = 2; arg < argc; ++arg)
    {
      switch (fnmatch (pattern, argv[arg], FNM_CASEFOLD|FNM_EXTMATCH))
        {
case FNM_NOMATCH:
  break;

case 0:
          printf ("\"%s\" matches \"%s\"\n", pattern, argv[arg]);
          ++count;
  break;

default:
  perror ("fnmatch");
  return 1;
        }
    }

  if (!count)
    printf ("\"%s\" does not match any of the other strings.\n");

  return 0;
}
```

Example 2.16: *Using fnmatch*

2.6.2 Filename Matching

```
$ rm *~
```

The '`*~`' in this example is a wildcard pattern, which describes the set of all strings

2.6.3 Regular Expression Matching

3 libstdc++ and the Standard Template Library

This chapter introduces '`libstdc++`'. Well, almost; '`libstdc++`' incorporates many things, one of which is the GNU implementation of the *Standard Template Library* (STL). So instead we'll be looking at a small, concentrated part of '`libstdc++`', in the form of the STL - justifiably so, given only one chapter. The Standard Template Library is very large and complex area of study; even reference books on it contain hundreds of pages. Therefore, this chapter looks at some of the more obvious uses of the STL, and detailed use is left for you to explore.

Why the STL? The STL is a large collection of useful programming utilities created to make programmers lives a lot easier. There are implementations of different containers that can hold data (such as lists, sets etc.), as well as generic algorithms that can be used with many of these containers. The STL is also standardised, meaning that wherever it is implemented, the interface and the results will be the same (unless the implementing parties didn't keep to the standard. . .). Also, the GNU project has worked hard at bringing it's implementation in accordance with the standard.

This chapter assumes that you already have knowledge of C++ and a fairly good understanding of object-oriented concepts. A good understanding of `templates` will also be useful.

Section 3.1 [How the STL is Structured], page 29 introduces the some of the basic components we'll be looking at, and how all these components work together. If you are new to STL, this is the place to start. We'll then look at practical use of containers (that store collections of objects) and iterators (used to traverse containers) in Section 3.2 [Containers and Iterators], page 31. We'll then look at Section 3.3 [Generic Algorithms and Function Objects], page 52 and see how we can combine algorithms and function objects with containers to provide a powerful set of programming tools to work with. Section 3.4 [Strings], page 64 shows us how STL provides an easy-to-use interface to strings, and how we can use STL strings with generic algorithms. A reference section is also provided, see Section 3.5 [STL Reference Section], page 68, giving a breakdown of all the commands used in this chapter as well as many more besides. Finally, Section 5.8 [Further Reading], page 125 provides a list of books and links for further reading.

The source code examples in this chapter all invoke `g++`, the GNU C++ compiler. Full details about compilation are given in Chapter 4 [The GNU Compiler Collection], page 75, although the examples given throughout this section will be easy enough for you not to have to worry about reading ahead. For example, the compilation command for the `vector1.cc` source file (given in Section 3.2.3 [Vector], page 37), is:

```
g++ vector1.cc -o vector1
```

This means that you should type this into the command prompt in the directory where '`vector1.cc`' is located, hit `<ENTER>`, and it will compile the source file '`vector1.cc`' and produce a binary named '`vector1`' (`-o vector1` means name the output file, or binary, '`vector1`'); you'd run the binary by typing

```
$ ./vector1
```

at the command prompt in the directory where '`vector1`' is located.

3.1 How the STL is Structured

If you are not already familiar with the general layout of STL, we'll look briefly here at the overall view of it without too much attention to detail.

The STL is made up of a number of different components: *containers*, *iterators*, *generic algorithms*, *function objects*, *adaptors* and *allocators*. Here's a brief description of each of them:

Containers Store collections of objects. Without being specific, think of any of the classic computing data structures: lists, sets: these are some of the containers of the STL. Containers can either store primitive data types - integers, characters etc. - or objects that we define ourselves.

Iterators Provide a means of traversing forwards (and possibly backwards) through containers. Think of them as objects that encapsulate pointers to the objects within a container: we create a container, like a list of characters, and then provide iterators to be able to traverse from beginning to end of the container, for example.

Generic algorithms Enable us to apply a number of different algorithms (such as sorting, replacing, performing calculations etc.) on the elements within containers, and for that matter, any data set. The whole idea of the STL is genericity and extensibility. Thus, given some algorithm - like finding an item in a container - we would expect to find a way of using the `find` method with any kind of object within a container. This is provided by STL generic algorithms, using iterators to manipulate and view the elements of the container.

Given our list of characters mentioned previously, we can now use an algorithm to sort, replace, reverse - in fact do anything that is computationally possible (...using C++!) in terms of an algorithm. We could use STL defined algorithms, or even provide our own. Even better (and this is the crux of the illustration) - we can use these algorithms on any container of strings, integers, objects...

Function objects Function objects enable us to use classes as functions by overloading the function operator, `operator()`. We can create many different instances of a function object (as you'd create many instances of an object), which means that we can maintain an internal state for each function object. If you're familiar with function pointers, the principle is pretty much the same. Moreover, we can also use function objects in association with generic algorithms, passing function objects as arguments to the algorithm.

Strings Provide a useful interface to using C++ strings. For a long time, C-like string handing has had a bad press usually to do with memory issues; class `string` provides a simple interface which guards us from the nasty low-level details. Strings encapsulate iterators just like containers, and so we can also use generic algorithms and function objects just like we can with containers.

Adaptors An adaptor lets us modify the interface to some STL component. For example, although we may be able to make a vector act like a stack (pushing and popping elements), it is still not 'a stack', it's a vector. Adaptors let us modify the interface to vector, so that we can make our own stack class by modifying the interface to vector.

Allocators Allocators describe the memory model used with your program, providing information about pointers, references, sizes of objects etc..

In fact, that isn't all there is to the STL. There's a lot more besides; there's also a lot of support for streams, but we do not have enough space here to talk about these things. In this chapter, due to space and time, we'll only look at containers, iterators, generic algorithms, function objects and strings - and even then briefly. If you feel like finding out more, the books given in Section 5.8 [Further Reading], page 125 contain a wealth of information.

This is a very brief (and raw) introduction to some of the components we'll be looking at. Let's move ahead and see them being used.

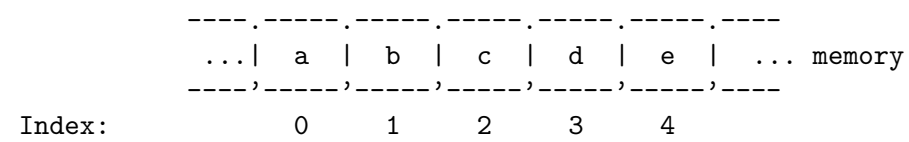
3.2 Containers and Iterators

Here we introduce the different containers available, as well as talk briefly about how iterators work with regard to containers. There are two groups of containers to be aware of: *sequence* and *sorted associative* containers (we'll look at iterators in a minute).

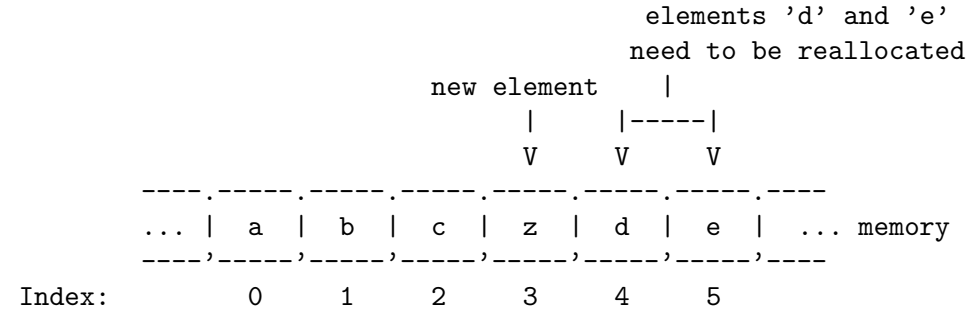
Sequence containers are exactly what they say - a sequence of elements. The elements are stored according to how they are inserted, or until you make some change (like performing a sort) to the ordering of elements. So creating a sequence container with ten elements means that when you access the first element, you are looking at the first element that was inserted; the second position is the second element you inserted, and so on.

Three sequence containers are provided: `vector`, `deque`[1] and `list`. `vector`s and `deque`s store elements contiguously in memory, so inserting elements can be expensive because (for example) if an element is inserted at the start of a vector, the remaining elements have to be reallocated. The advantage of this is that elements can be accessed very quickly via their index (like an array).

For example, consider the following `vector` containing characters 'a' through 'e':



Suppose that we want to insert a new character, 'z', between elements 2 and 3; to do so would involve reallocating memory from element 3 onwards to cope with the new insertion because elements are stored contiguously. This takes time of course; the elements that sit to the right of the newly inserted element need to be reallocated. Thus, the more elements you have, the longer it takes to insert new elements near the start of the `vector`. Inserting the new character is represented diagrammatically below:



...and because the elements have been reallocated ('d' is now located at index 4 etc.), we can retrieve them via their index in constant time. Deleting elements is also costly, for the same reason - if you take out element 3 ('z'), 'd' and 'e' need to be reallocated so that they occupy their old positions again. As compensation however, if we want element 3, we know exactly where it is (just like using an array), so although it may take time to insert and remove elements, retrieval is fast.

TODO: deque explanation w. diagrams etc.

`list`s on the other hand provide quick insertion and deletion times because elements are not stored contiguously in memory, but as a consequence you cannot access elements via an index - you have to move through the list looking at each element to determine what it contains.

[1] *deque* stands for *double-ended queue* and is pronounced *deck* as in 'deck of cards'.

TODO: list explanation w. diagrams etc.

Sorted associative containers on the other hand are stored via *keys* which aim to make retrieval quicker than sequence containers. You do not access them by their location (as you would a sequence container), but instead access them by their key. So if you had a sorted associative container with the key being a string and the value being an integer (for example a *name*:*phone-number* pairing), you would retrive the integer value by accessing the relevant key. For example if you had an element that had the key `"John Smith"` and the value 456123, you get the number by asking for the element with key `"John Smith"`.

There are four sorted associative containers available, split into two categories: *set* and *map*. With `set`s, the data items are the keys themselves; `map`s on the other hand store data as a *key*:*value* pairing. `set` and `map` allow unique keys only; `multiset` and `multimap` allow duplicate keys.

TODO: set explanation w. diagrams etc.

TODO: map explanation w. diagrams etc.

Before we proceed, a note about time complexities. In the following table, we use the term O to represent *big-oh* notation. That is, we use O to represent the worst-case scenario for some computation. For example, if you have a dynamic array of characters and you want to search for an element in the array, the time complexity for the search will be $O(n)$ - or linear time - in otherwords, we know that we'll not have to make more than n tests to find some arbitrary element. The different concepts that we'll refer to at various points in this chapter involve looking at *constant*, *linear* and *logarithmic* time. Here's a brief explanation of each one:

Time	**Discussion**
Constant	The best time we can hope for and acheive; constant time operations are performed in $O(1)$ time.
Linear	Achieved in $O(n)$ time - for example, searching an unordered linked list for an element would take linear time.
Logarithmic	Operations are performed in $O(logn)$ time. This is an improvment on linear time operations, and a classic example would be a binary tree.

Now time complexity is out of the way, the following table summarises the different containers available along with their benefits:

Container	**Insertion Time**	**Deletion Time**	**Reference Invalidation**[2]	**Retrieval Time**
`vector`	For elements at the end: O(n) if there is no space for the element, and all elements need to be reallocated; O(1) otherwise. For all other elements: O(n)	For elements at the end - constant O(1) time; for all other elements: O(n)	Yes	Constant
`deque`	For elements at the beginning and end: O(n) if there is no space for the element, and all elements need to be reallocated; O(1) otherwise. For all other elements: O(n).	For elements at the beginning and end: O(1). For all other elements: O(n).	Yes	Constant

`list`	Constant: elements can be inserted anywhere in O(1) time.	Constant: elements can be deleted anywhere in O(1) time.	No	O(n)
`set`, `multiset`	O(log n)	Where there are *i* elements to be deleted, the time complexity is O(log n+i)	No	O(log n)
`map`, `multimap`	O(log n)	Where there are *i* elements to be deleted, the time complexity is O(log n+i)	No	O(log n)

Before moving on, let's look at some real-world examples of practical applications that would use STL containers.

Application	**Container(s)**	**Explanation**
Telephone Directory	`map`, `multimap`	A simple telephone directory would contain an address with a corresponding telephone number. You'd search the directory for an address, and the address would give you the phone number. Simple, really, but the important reason for using `map` is that it enables us to search in logarithmic time. For a telephone directory of many entries, this is very important: a directory holding 16 million entries would require no more than 24 steps.
Order Processing System	`vector`, `deque`	Consider a simple order processing system where orders are received and placed in some kind of queue. `vector` and `deque` are perfect for the job: they enable us to store elements contiguously and as a result, we can model the way in which orders are received by queuing them. `deque` would probably be our best solution: we add order placements to the end of the `deque` in the order that they arrive; and we remove them from the front, like a FIFO (first-in, first-out) queue.

The following sections provide programming examples for each of these containers. Depending on what problem you are working with, different containers will provide different advantages. Just for the record, STL provides a number of container adaptors, namely `stack`, `queue` and `priority_queue` - the names of which should give you some idea of what they do. A special container is also provided called `bitset`. However, none of these containers are dealt with in this chapter; see Section 5.8 [Further Reading], page 125 for reference material.

3.2.1 Preliminaries

Throughout this chapter we'll focus on using objects as elements of containers. Instead of demonstrating the different containers and algorithms (etc.) using primitive data types, we'll instead use a simple C++ class (although, initially with `vector`, we'll begin with primitive data types).

The examples will revolve around using an address class - albeit very simple (and unrealistic!). Here's the header file, '`Address.hh`':

```
/* Address.hh */
#ifndef Address_hh
#define Address_hh
```

```
#include <string>

class Address
{
public:
  Address(){name=street=city=""; phone=0;}
  Address(string n, string s, string c, long p);
  void print() const;
  bool operator < (const Address& addr) const;
  bool operator == (const Address& addr) const;
  string getName() const {return name;}
  string getStreet() const {return street;}
  string getCity() const {return city;}
  long getPhone() const {return phone;}
private:
  string name;
  string street;
  string city;
  long phone;
};

#endif
```

Example 3.1: *Address.hh*

. . . and the definition, '`Address.cc`':

```
/* Address.cc */
#include "Address.hh"

Address::Address(string n, string s, string c, long p)
{
  name = n;
  street = s;
  city = c;
  phone = p;
};
void Address::print() const
{
  cout <<
    "Name: " << name << ", " <<
    "Street: " << street << ", " <<
    "City: " << city << ", " <<
    "Phone: " << phone << endl;
};
bool Address::operator < (const Address& addr) const
{
  if (name < addr.getName())
    return true;
  else
    return false;
};
bool Address::operator == (const Address& addr) const
```

```
{
  if (name == addr.getName())
    return true;
  else
    return false;
};
```

Example 3.2: *Address.cc*

We've added operators because they'll be useful later when we come to sort the elements using different containers. The equality operators are not very strict; we're only interested in comparing names, and consider that two people with the same name to the same person; this isn't really important, given the limited nature of the examples to follow.

In addition, we'll be using a header file to keep a number of different `Address` objects in:

```
/* AddressRepository.hh */
#ifndef Address_Repoditory_hh
#define Address_Repoditory_hh

#include "Address.hh"

Address addr1("Jane", "12 Small St.", "Worcs", 225343);
Address addr2("Edith", "91 Glib Terrace", "Shrops", 858976);
Address addr3("Adam", "23 Big St.", "Worcs", 443098);
Address addr4("Jane", "55 Almond Terrace", "Worcs", 242783);
Address addr5("Bob", "2 St. Annes Walk", "Oxford", 303022);

#endif
```

Example 3.3: *AddressRepository.hh*

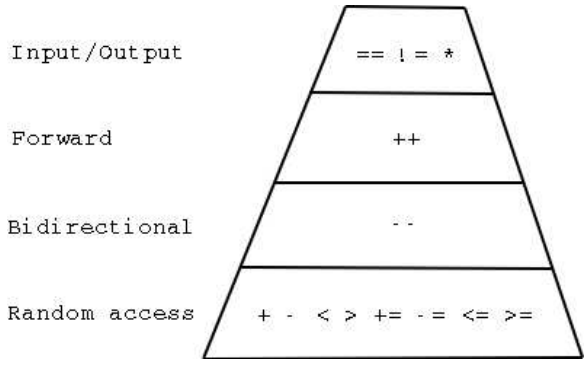
3.2.2 A Crash Course in Iterators

Before continuing with containers, we'd better pause briefly to explain *iterators*. Iterators are important because they allow us to move through elements of a container. There are a few kinds of iterators, so we'll discuss each of them in turn briefly now - the purpose here is to merely describe the different kinds of iterators available, to prepare you for the succeeding sections.

It is very important to understand the capabilities of each iterator because each container uses a certain type of iterator, and generic algorithms require certain iterators. Because containers can be used with generic algorithms to search or replace an element (for example), it is important to be able to distinguish *which* iterators can be used with which algorithms (more on this later - Section 3.3.4 [Introducing Generic Algorithms], page 57).

An iterator is a smart pointer; it enables us to keep track of where we are in a container. We can use operators, like `++` to move forward one element, as well as the usual operators that we'd use for pointer arithmetic. Well, almost; the kind of iterator we're using determines exactly what operators can be used. There is a hierarchy of iterators to be aware of that will carry us through the rest of this chapter - certain containers and algorithms have to use certain iterators, which we have to be strict about to avoid compile errors.

The different iterators available are described hierarchically below in figure (**FIXME:** *fig. ref. to do.*):



Each level possesses all the abilities of any iterators that are above it; so bidirectional iterators posses the abilities of forward and input/output iterators, whereas random access iterators possess all of the abilities of input/output, forward and bidirectional iterators, as well as its own operators.

Let's look at these each in turn:

Input iterators read elements they encounter element by element. Input iterators can read an element *only once*, thus if you attempt to reread the same position, it is not gauranteed that you'll read the same element. A good example of an input iterator would be reading characters from a stream, like when you read characters from the keyboard.

Similar to input iterators, output iterators write elements out element by element, and you cannot re-iterate over the same range of elements once you have started traversing them. Again, a good example of an output iterator would involve writing characters out to a stream - for example writing characters from the keyboard to standard output.

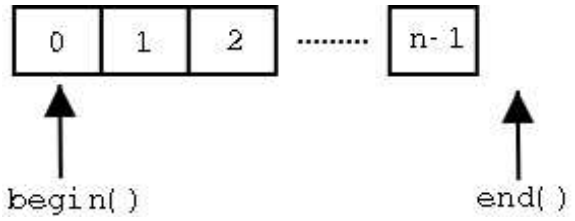
Forward iterators, as well as having all of the operations of input iterators and some of the operations of output iterators, can also refer to the same element more than once. Thus, you could traverse the range of elements from start to end, and then re-iterate over that range - for example finding an element in a vector. We can search for the first occurence of some value, and then search the container again starting from where we left off.

As well as having the properties of forward (and therefore, input and output) iterators, bidirectional iterators can also move *backwards* through a range of elements. Thus instead of being restricted to only searching forwards (as in the previous example), we can also step backwards through the container of elements.

Random access iterators have all the properties of bidirectional iterators (and therefore all of the properties of forward and input/output iterators), but can also access elements without having to traverse them in an element by element fashion. In addition, <, <=, > and >= operators are also provided for. Random access iterators are very powerful because they allow to do things like make binary searches.

A note about iterators and containers. Containers provide two member functions, `begin()` and `end()`, to enable us to see the start and end of the range. However, whilst `begin()` points

to the first element of a container (if there exists at least one element), `end()` points past the *last* element of the container. This is represented diagramatically below for n elements:



Thus, for any container - `vector`, `map`, `set`, etc., providing there is at least one element, you can be assured `begin()` will point to the first, and `end()` will point past the last element. The reason for `end()` pointing past the *last* element is practicality; for example, the `find` algorithm returns an iterator to an object in the container (if it found one), else it returns `end()`, thus enabling us to check to see if the find failed or not (that is, if `end()` is returned, the element we're looking for does not exist within that container).

This concludes our (very) brief tour of iterators. Reference texts contain much more information (see Section 3.5 [STL Reference Section], page 68), although we'll not look any further at iterators. The following sections explore the various containers available, which make use of iterators.

3.2.3 Vector

A vector is like a dynamic array. Thus, you can add elements to the vector, and access them in constant time - being able to access each element as you need. Insertion and deletion of elements at the start or middle of the vector takes linear time; elements at the end take constant time. Think of a vector as a dynamic array of elements; you can change the size as you see fit and insert and retrieve elements according to an index. Inserting and deleting elements anywhere other than the end is costly in terms of time because the elements need to be reallocated - because container elements are stored contiguously in memory, inserting and deleting elements means that succeeding elements need to be reallocated. To use a vector in your program, you must include `<vector>`.

Let's take a look at a simple source file that uses a vector:

```
/* vector1.cc
   Compiled using g++ vector1.cc -o vector1 */
#include <vector>

int main()
{
  std::vector<int> v;
  for (int i=0; i<10; i++)
    v.push_back(i);
  cout << "v now contains " << v.size() << " elements:" << endl;
  for (int i=0; i<v.size(); i++)
    cout << "'" << v[i] << "' ";
  cout << endl;
  exit(0);
}
```

Example 3.4: *vector1.cc*

which produces the following output:

```
$ ./vector1
v now contains 10 elements:
'0' '1' '2' '3' '4' '5' '6' '7' '8' '9'
$
```

First, we declare that we'll be using a vector with the declaration `#include <vector>`. The declaration `std::vector<int> v` creates an empty vector named `v` with no elements whatsoever, which will contain elements of type `int`. We then iterate through a loop ten times, using the `push_back(elem)` method.

`push_back` is an efficient means of insertion for `vector` - each element is pushed onto the back of the vector without any need of reallocating previous elements, which as we've already mentioned can be costly.

We then utilize the `size()` method to print the amount of elements the vector contains, and then loop through the elements from `0` to `v.size()-1`, using the array subscript operator. Note that we can use array subscript operators to access elements of a vector, and thus avoid the use of iterators.

That's OK, but we can rewrite the above example to produce the same result but using essentailly different code and making use of iterators:

```
/* vector2.cc
   Comiled using g++ vector2.cc -o vector2 */
#include <vector>

int main()
{
  std::vector<int> v(10);
  /* Declare an iterator to work with: */
  std::vector<int>::iterator pos;
  cout << "v now contains " << v.size() << " elements:" << endl;
  int counter = 10;
  for (pos = v.begin(); pos != v.end(); ++pos)
    {
      --counter;
      *pos = counter;
    }
  pos = v.begin();
  while(pos != v.end())
    {
      cout << *pos << ' ';
      ++pos;
    }
  exit(0);
}
```

Example 3.5: *vector2.cc*

The output to this is obvious: we inform the user that the `vector` contains 10 elements, and then print them out one by one which involves printing out the numbers 9 down to 0.

This time we opt to declare a vector named `v`, reserving 10 elements of type int with the declaration `std::vector<int> v(10)`. In addition, we also declare an iterator to work by declaring `std::vector<int>::iterator pos`. We couldn't have just declared an iterator on it's own, like `iterator pos`, for example; the reason being is that each container has it's own iterator (and as

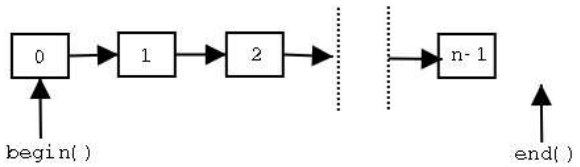
a consequence we didn't need to declare `#include <iterator>` because `vector` already includes it). Let's look at how the iterator works.

Recall from Section 3.2.2 [A Crash Course in Iterators], page 35 that there are many kinds of iterators, suited to different containers. Vector uses a *random access* iterator; this contains all of the properties of a bidirectional iterator, as well as being able to use random access. Thus, we can use all of the comparison operators with `pos`, as well as pre- and post-increment operators.

An iterator enables us to maintain track of where we are in a container much the same way we use pointer arithmetic. Therefore, if we set `pos` at the beginning of a container of objects, we'd expect `++pos` to point to the next element (if it exists...), and `pos += 3` to point to the 3rd element after the current object. Therefore, the `for` loop

```
for (pos = v.begin(); pos != v.end(); ++pos)
{
        --counter;
        *pos = counter;
}
```

takes advantage of the `begin()` and `end()` methods of vector. The `begin()` method allows us to access the *first* element of the vector; the `end()` method points *past* the last element of the vector. Each of these methods return an iterator. This is represented in figure (TODO: figure no.):



The `++pos` statement in the `for` loop sets `pos` to the next element in the container using the pre-increment operator. Using pre-increment generally offers better performance - using `pos++` returns the *old* position of the iterator, whereas `++pos` returns the *new* position of the iterator. The decision to use a `while` loop *and* a `for` loop in the example was arbitrary; it was merely to demonstrate the different ways in which iterators could be used with a loop. Declaring a vector, adding elements and then reading them back are very simple, so let's look at some other vector operations we can perform. The following example illustrates how we can use different ways to access elements as well as some other methods of inserting and removing elements. We'll concentrate on using objects as elements rather than primitive data types:

```
/* vector3.cc
 * Compiled using g++ vector3.cc Address.cc -o vector3 */
#include <vector>
#include "AddressRepository.hh"
int main()
{
  vector<Address> v;
  /* Add all of the address objects to the vector: */
  v.push_back(addr1);
  v.push_back(addr2);
  v.push_back(addr3);
  v.push_back(addr4);
  v.push_back(addr5);
  /* Declare an iterator to work with: */
  std::vector<Address>::iterator pos;
  /* Loop through the vector printing out elements: */
```

```
  cout << "First iteration" << endl;
  for (pos=v.begin(); pos<v.end(); ++pos)
    {
      pos->print();
    }

  /* Remove the last element: */
  v.pop_back();
  /* Create and insert a new Address object: */
  Address addr6("Reggie", "1 Card Rd.", "Hamps", 892286);
  v.insert(v.begin(), addr6);
  cout << "Second iteration" << endl;
  for (pos=v.begin(); pos<v.end(); ++pos)
    {
      pos->print();
    }
  exit(0);
}
```

Example 3.6: *vector3.cc*

and when compiled and run produces the following output:

```
$ ./vector3
First iteration
Name: Jane, Street: 12 Small St., City: Worcs, Phone: 225343
Name: Edith, Street: 91 Glib Terrace, City: Shrops, Phone: 858976
Name: Adam, Street: 23 Big St., City: Worcs, Phone: 443098
Name: Jane, Street: 55 Almond Terrace, City: Worcs, Phone: 242783
Name: Bob, Street: 2 St. Annes Walk, City: Oxford, Phone: 303022
Second iteration
Name: Reggie, Street: 1 Card Rd., City: Hamps, Phone: 892286
Name: Jane, Street: 12 Small St., City: Worcs, Phone: 225343
Name: Edith, Street: 91 Glib Terrace, City: Shrops, Phone: 858976
Name: Adam, Street: 23 Big St., City: Worcs, Phone: 443098
Name: Jane, Street: 55 Almond Terrace, City: Worcs, Phone: 242783
```

This example isn't too different from the last, although there are a few important points to make.

The `push_back` method inserts elements at the end of the vector; this method of insertion is extremely fast and should be preferred if you are looking for fast insertion - none of the elements need to be reallocated. Notice that we declare the iterator to have type `Address`, rather than `int` as in the previous examples. Since `pos` is an `iterator`, and is a smart pointer to the containers element under scrutiny, we can simply access the `Address` objecct directly with a call to `pos->print()`, because `pos` points to each element in the `vector`.

After adding 5 `Address` objects, `v.pop_back()` removes the last element of the vector; so at this point in the execution of the binary, there will only be four elements in the vector, since Edith - the last element in the vector - has been removed. Again, removing the *last* element of a `vector` is also fast and achieved in constant time.

We also insert an element into the vector using the line `v.insert(v.begin(), addr6)`. Inserting `addr6` at `v.begin()` results in `addr6` being inserted at the start of the vector. This is costly in terms of time; all of the remaining elements need to be reallocated after inserting an

element at the start of the vector. Finally, the second loop prints out the vector of `Address` objects.

As you can see the interface to using vector is very simple, and the difference between using primitive datatypes and objects as vector elements is trivial.

There are a few points to make before we finish and move on to `deque`. We haven't dealt with deletion yet, and there are a number of consequences when deleting elements from a `vector`. We'll also focus briefly on reallocation and capacity of `vector`.

Since `vector` stores elements contiguously, if we delete an element, an important consequence follows: that all previously assigned references, iterators and pointers to any *succeeding* elements in the `vector` are *invalidated*. By invalidated, we're really saying that, "this (pointer/reference/iterator) is no longer reliable". Let's look at a simple example.

```
/* vector 4.cc
   Compiled using g++ vector4.cc -o vector4 */
#include <vector>
#include "AddressRepository.hh"

int main()
{
  int *ill_ptr;

  std::vector<int> v;
  for (int i=0; i<10; i++)
    v.push_back(i*100);

  std::vector<int>::iterator pos = v.begin();
  pos += 5;
  ill_ptr = pos;

  cout << "Element 5: " << *ill_ptr << endl;
  /* Erase the first element: */
  v.erase(v.begin());
  /* Now print out the old 'pos': */
  cout << "... after reallocation: " << *ill_ptr << endl;
  exit(0);
}
```

Example 3.7: *vector4.cc*

The output of the program is fairly predictable:

```
Element 5: 500
... after reallocation: 600
```

The above program pushes ten integers onto the `vector`, so that it stores the values 0, 100, 200 and so on up to 900. At the point where we find the element `500`, we assign `ill_ptr` to the `iterator` already pointing to `500`. The result is that we perform `v.erase(v.begin())` to delete the first element, all of the elements of the `vector` are reallocated. Consequently, instead of `ill_ptr` pointing to the value `500`, it instead contains `600`. Although the address of the pointer points to the same location, the *contents* of where the pointer points to has changed due to the reallocation, and has been invalidated.

Another point to be aware of when using `vector` is capacity and reallocation. So far `vector` seems to lack because of the issues of inserting and deleting elements, which can be time consuming if you insert or delete any elements other than the *last* element. If speed is an important

factor, then we need to avoid reallocation where necessary because reallocation takes time. OK; so is there a way around this? Luckily there is, and it is down to the `capacity` and `reserve` methods.

`capacity` tells us how many elements we could place in a `vector`. This is pretty useful; it means that we can create a `vector` and tell it how much space to reserve for us, using the `reserve` method (see Section 3.5.1 [Container Summary], page 68 for details). Because space has been stored for the elements (providing a constructor is provided for the elements we're inserting), when we come to `insert` elements, no reallocation is necessary *unless* the capacity is exceeded. The `reserve(n)` method ensures that we can create a `vector` with *at least* `n` elements. Providing your objects have a default constructor, you could just call `std::vector<Type> v(n)`, where `Type` is the data type, and `n` is the number of elements you wish to create using the default constructor of data type `Type`. Extra time will have to be taken to instantiate the objects however, so `reserve()` will probably a better option.

There are too many methods to illustrate using just examples, and the preceeding examples are simple enough for you to be able to use `vector` on a basic level. All of the available member functions of `vector` are detailed in Section 3.5.1 [Container Summary], page 68. More complex examples will be seen when we explore Section 3.3 [Generic Algorithms and Function Objects], page 52.

3.2.4 Deque

`deque`[3] provides the same functionality as do `vector`s (linear time insertion and deletion in the middle of the container, constant time insertion and deletion at the end), and in addition provide constant time insertion and deletion of elements at the start of the `deque`. To use a deque in you program, you'll have to include `<deque>`.

This is the only advantage that `deque` offers over `vector`. You should only use them if you will definately be performing regular insertions or deletions at the start and end, and time is an important factor for such modifications.

There is little other difference between a `vector` and a `deque`, other than performance. The following example is just a re-work of the example given for `vector3.cc`; the only difference is that all occurrences of `vector` are replaced with `deque`, and we've renamed `vector<Address v>` to `deque<Address> d`.

```
/* deque1.cc
   Compiled using g++ deque1.cc Address.cc -o deque1 */
#include <deque>
#include "AddressRepository.hh"
int main()
{
  deque<Address> d;
  /* Add all of the address objects to the deque: */
  d.push_front(addr1);
  d.push_front(addr2);
  d.push_front(addr3);
  d.push_front(addr4);
  d.push_front(addr5);
  /* Declare an iterator to work with: */
  std::deque<Address>::iterator pos;
  /* Loop through the deque printing out elements: */
  cout << "First iteration" << endl;
```

[3] Short for "double-ended queue" and pronounced *deck* as in deck of cards.

```
  for (pos=d.begin(); pos<d.end(); ++pos)
    {
      pos->print();
    }

  /* Remove the first element: */
  d.pop_front();
  /* Create and insert a new Address object: */
  Address addr6("Reggie", "1 Card Rd.", "Hamps", 892286);
  d.pop_front(addr6);
  cout << "second iteration" << endl;
  for (pos=d.begin() ; pos<d.end(); ++pos)
    {
      pos->print();
    }
  exit(0);
}
```

Example 3.8: *deque1.cc*

The output's a little different, because we used `push_front` and `pop_front` instead of `push_back` and `pop_back` respectively:

```
$ ./deque1
First iteration
Name: Bob, Street: 2 St. Annes Walk, City: Oxford, Phone: 303022
Name: Jane, Street: 55 Almond Terrace, City: Worcs, Phone: 242783
Name: Adam, Street: 23 Big St., City: Worcs, Phone: 443098
Name: Edith, Street: 91 Glib Terrace, City: Shrops, Phone: 858976
Name: Jane, Street: 12 Small St., City: Worcs, Phone: 225343
second iteration
Name: Reggie, Street: 1 Card Rd., City: Hamps, Phone: 892286
Name: Jane, Street: 55 Almond Terrace, City: Worcs, Phone: 242783
Name: Adam, Street: 23 Big St., City: Worcs, Phone: 443098
Name: Edith, Street: 91 Glib Terrace, City: Shrops, Phone: 858976
Name: Jane, Street: 12 Small St., City: Worcs, Phone: 225343
```

You'll agree that the code isn't to different from '`vector3.cc`', but we've already mentioned that `vector` and `deque` are similar. Let's look at a few of the main differences between these two containers.

Inserting elements at the front of a `deque` is the primary advantage over `vector`. In the example above, we've demonstrated this by using the `push_front` member function. By contrast however, performing `push_front` operation with a `vector` would have been very costly.

Insertions at the beginning or end of a `deque` invalidate all (previously allocated) pointers, iterators and references. By contrast, `vector` pointers, iterators and references are invalidated anytime an element with a smaller index is inserted or removed, or the capacity changes due to reallocation. You should consider this carefully if you're going to use a `deque`; reallocating memory every time you make a non-ended insertion or deletion could seriously slow things up, especially for large collections of elements.

Generally, the similarities between `vector` and `deque` mean that you should really consider using a `deque` over `vector` when elements will be added and removed at both ends of the container, with little insertion or removal in the middle.

3.2.5 List

The `list` container is a very different sequence container compared to `vector` and `deque`. `list` does not use random access iterators, but as a trade-off allow constant time insertion and deletion at *any* point in the list, instead of reallocating memory to cope with the locations of elements in the container. Elements are just inserted into the list by providing a link from the element it was inserted after and a link to the element it was inserted before. It is much like a naive linked-list data-structure in which each time elements are added, instead of reordering the data, you just slip them in to wherever they need to be, ignoring the overall structure and order of the list. The same is true of element deletion. The consequence of this is that we can insert or remove elements quickly, but as a forfeit sacrafice the ability to use random access iterators (we've given up the ability to recall where element `i` is). Lists are included in your program by including `<list>`.

With no need to worry about reallocation of elements within the list, insertions do not invalidate iterators, and deletions only invalidate elements which are being referred to.

The lack of random access also means that a few key generic algorithms will not work. These are instead defined as member functions of `list`.

Let's look at a few examples of using a list.

```
/* list1.cc
   Compiled using g++ list1.cc Address.cc -o list1 */
#include <list>
#include "AddressRepository.hh"
int main()
{
  list<Address> list1;
  /* Add all of the address objects to the list: */
  list1.push_front(addr1);
  list1.push_front(addr2);
  list1.push_front(addr3);
  list1.push_front(addr4);
  list1.push_front(addr5);
  /* Declare an iterator to work with: */
  std::list<Address>::iterator pos;
  /* Loop through the list printing out elements: */
  cout << "Iterating through list1: " << endl;
  for (pos=list1.begin() ; pos!=list1.end(); ++pos)
    {
      pos->print();
    }
  /* Create a new list to work with: */
  list<Address> list2;
  for (pos=list1.begin(); pos!=list1.end(); ++pos)
    {
      list2.insert(list2.begin(), *pos);
    }

  /* Create and insert a new Address object: */
  Address addr6("Reggie", "1 Card Rd.", "Hamps", 892286);
  list2.push_front(addr6);
  cout << "Iterating through list2:" << endl;
  for (pos=list2.begin() ; pos!=list2.end(); ++pos)
```

```
    {
      pos->print();
    }
  exit(0);
}
```

Example 3.9: *list1.cc*

A few changes have been made since the `deque` and `vector` examples that also utilised the `Address` class. However, it's been modified to create a copy of list `list1` using the `insert` function to add elements to `list2`:

```
list<Address> list2;
for (pos=list1.begin(); pos!=list1.end(); ++pos)
  {
    list2.insert(list2.begin(), *pos);
  }
```

This doesn't really demonstrate anything unless we actually time it; recall that inserting elements into a `list` is extremely fast. Thus, whereas with a `vector` or `deque` we can use `insert` as needed, it is extremely inefficient becuase memory needs to be reallocated each time. Our `list` does not suffer from this restriction, so we insert elements *at will* knowing that it will be fast.

Using algorithms to sort data in containers is discussed in Section 3.3 [Generic Algorithms and Function Objects], page 52. However, these sorting algorithms will not work with `list` because we need random access iterators to be able to sort data. Therefore, a number of member functions are defined that enable us to take advantage of a number of algorithms denied to us. These are `sort` to sort data using the less-than equality operator and `unique` to remove duplicate elements in a list (Section 3.3.2 [Some Predefined Function Objects], page 55 discusses how you can alter the default sorting criterion on a list). They're trivial to use, as can be seen in the example below:

```
/* list2.cc
   Compiled using g++ list2.cc Address.cc -o list2 */
#include <list>
#include "AddressRepository.hh"
int main()
{
  list<Address> list1;
  /* Add all of the address objects to the list: */
  list1.push_front(addr1);
  list1.push_front(addr2);
  list1.push_front(addr3);
  list1.push_front(addr4);
  list1.push_front(addr5);

  /* Sort the list: */
  list1.sort();
  /* Remove any duplicate names: */
  list1.unique();
  std::list<Address>::iterator pos;
  for (pos=list1.begin() ; pos!=list1.end(); ++pos)
    {
      pos->print();
```

```
    }
  exit(0);
}
```

Example 3.10: *list2.cc*

The result is fairly predictable; the elements are sorted according to the < operator, and then any unique elements are removed - in this instance, `Address` objects that have matching names "Jane". Here's the output:

```
$ ./list2
Name: Adam, Street: 23 Big St., City: Worcs, Phone: 443098
Name: Bob, Street: 2 St. Annes Walk, City: Oxford, Phone: 303022
Name: Edith, Street: 91 Glib Terrace, City: Shrops, Phone: 858976
Name: Jane, Street: 55 Almond Terrace, City: Worcs, Phone: 242783
```

3.2.6 Set

Sets enable you to declare store data collections as individual items, although no duplicate elements are allowed (all elements must be unique). Unlike our previous examples of sequence containers (`vector`, `deque` and `list`), we can retrieve elements from a `set` (as we can from all sorted associative containers) rapidly - in logaritmic time - in comparisson with sequence containers, which are much slower. You use a set in your code by including `<set>`.

Inserting elements into a set is a little different to normal; a `pair` object is returned from an insertion, the first parameter being an iterator and the second being a boolean value, which determines if there were any duplicates or not. If there were, then the element is not inserted (only unique elements are allowed). Here's a simple example:

```
/* set1.cc
   Compiled using g++ set1.cc Address.cc -o set1 */
#include <set>
#include "AddressRepository.hh"

int main()
{
  set<Address> set1;
  if (!set1.insert(addr1).second)
        cout << "Failed to insert addr1" << endl;
  if (!set1.insert(addr2).second)
        cout << "Failed to insert addr2" << endl;
  if (!set1.insert(addr3).second)
        cout << "Failed to insert addr3 " << endl;
  if (!set1.insert(addr4).second)
        cout << "Failed to insert addr4 " << endl;

  std::multiset<Address>::iterator pos;
  cout << "set1 now contains: " << endl;
  for (pos=set1.begin(); pos!=set1.end(); ++pos)
    {
      pos->print();
    }
  exit(0);
}
```

Example 3.11: *set1.cc*

The program produces the following output:

```
$ ./set1
Failed to insert addr4
set1 now contains:
Name: Adam, Street: 23 Big St., City: Worcs, Phone: 443098
Name: Edith, Street: 91 Glib Terrace, City: Shrops, Phone: 858976
Name: Jane, Street: 12 Small St., City: Worcs, Phone: 225343
```

Notice that elements are printed ascending alphabetically. This is because the default sorting criterion for a set uses < (Section 3.3.2 [Some Predefined Function Objects], page 55 discusses how to change this ordering). Obviously, the `Address` object with the name Jane is not inserted. It's not because `addr1` and `addr4` have identical values such as street address, phone number etc.; this is incidental. Recall from Section 3.2.1 [Preliminaries], page 33 that we defined a < operator for class `Address`, which tests only for names. Thus, when `addr4` is about to be inserted, the insert operation finds that `"Jane"` already exists within the set and as a consequence `addr4` is not inserted. The `pair` objects values can be accessed using the member variables `first` and `second`. As mentioned previously, when used with `insert`, `first` returns the iterator from the inserted element, and `second` returns whether the element was inserted or not.

Like `list`, `set` provides a number of searching methods that enable you to perform logarithmic-time complexity operations. This is in preference to the linear-time complexity of the searching algorithms provided by generic algorithms (see Section 3.3 [Generic Algorithms and Function Objects], page 52).

We'll only use a few searching methods with `set`; the others are easier used with `multiset`.

```
/* set2.cc
   Compiled using g++ set2.cc Address.cc -o set2 */
#include <set>
#include "AddressRepository.hh"

int main()
{
  set<Address> set2;
  set2.insert(addr1);
  set2.insert(addr2);
  set2.insert(addr3);
  set2.insert(addr4);
  set2.insert(addr5);

  std::set<Address>::iterator pos;
  cout << "Found " << set2.count(addr4) << " elements with name \""
       << addr4.getName() << "\"" << endl;
  pos = set2.find(addr4);
  if (pos != set2.end())
    {
      cout << "Found: ";
      pos->print();
    }
  else
    {
      cout << "Could not find ";
```

```
      pos->print();
    }
   exit(0);
}
```

Example 3.12: *set2.cc*

Notice to start with that `count` takes an element of the set as an argument; it looks through the set and counts how many occurrences there are of that element. So `set2.count(addr4)` returns how many occurrences of `addr4` there are in `set2`. Obviously in our case we'll only find one, because sets do not allow duplicates. Finding the element `addr4` is easy, and can be achieved in logarithmic time using `find`. Although trivial for this example, it is extremely useful for much larger collections of objects to be able to make searches so quickly. `find` takes an element as an argument and returns an `iterator` to it. So in the section of code

```
pos = set2.find(addr4);
Address addr;
if (pos != set2.end())
  {
    cout << "Found: ";
    pos->print();
  }
```

we first use an iterator to be assigned the location of the result of the `find` method (a failure returns `end()`) in the call `pos = set2.find(addr4)`, and then if `pos` isn't equal to `set2.end()`, we print out the result of the find.

Here's the output:

```
$ ./set2
Found 1 element with name Jane
Found: Name: Jane, Street: 12 Small St., City: Worcs, Phone: 225343
```

3.2.7 Multiset

Since the only difference between `set` and `multiset` is whether or not duplicates are allowed, we'll focus only for a moment about the implications this brings to `multiset`s. Like `set`, you use a `multiset` by including `<set>`.

Because elements are ordered by their key with sorted associative containers, we need someway of dealing with duplicate elements regarding `multiset`. The concept of looking for duplicate elements is fairly simple; to look at the first element of a set of duplicates, we make a call to `lower_bound(e)`, which finds the *first* occurrence of element `e`. Finding the upper bound is done by calling `upper_bound(e)`, which points past the *last* occurrence of element `e`[4]. Both methods return an `iterator` to the position of element `e` with some multiset `m`. Let's take a look at some examples of using a multi set:

```
/* multiset1.cc
   Compiled using g++ multiset1.cc Address.cc -o multiset1 */
#include <set>
#include "AddressRepository.hh"

int main()
{
  multiset<Address> mset1;
```

[4] Much the same as the `end` function points past the last element as described in Section 3.2.3 [Vector], page 37

```
  mset1.insert(addr1);
  mset1.insert(addr2);
  mset1.insert(addr3);
  mset1.insert(addr4);
  mset1.insert(addr5);

  Address addr("Jane", "33 Trimpley Close", "Kidderminster", 997331);
  mset1.insert(addr);
  std::multiset<Address>::iterator pos;
  cout << "Found " << mset1.count(addr4) << " elements with name "
       << addr4.getName() << endl;
  for(pos = mset1.lower_bound(addr4);
      pos != mset1.upper_bound(addr4);
      ++pos)
    {
      cout << "Found: ";
      pos->print();
    }
  exit(0);
}
```

Example 3.13: *multiset1.cc*

This example isn't really different from `set2.cc`, except that we're now using `upper_bound` and `lower_bound`, two functions which are only useful when we're dealing with duplicate elements. `Address` objects `addr1` and `addr4` are considered duplicates because the names are the same, and we've added a new `Address` object to be a duplicate with `addr1` and `addr4` also. The net result is that we end up with a `multiset` with 6 `Address` objects, three of them duplicates with the `name` variable set to `"Jane"`.

The section

```
  for(pos = mset1.lower_bound(addr4);
      pos != mset1.upper_bound(addr4);
      ++pos)
    {
      cout << "Found: ";
      pos->print();
    }
```

enables us to loop move through the `multiset`, from the first occurence of an `Address` object with the name `"Jane"`, to the last occurrence of an object with the name of `"Jane"`. Once again, the output is fairly obvious:

```
Found 3 elements with name Jane
Found: Name: Jane, Street: 12 Small St., City: Worcs, Phone: 225343
Found: Name: Jane, Street: 55 Almond Terrace, City: Worcs, Phone: 242783
Found: Name: Jane, Street: 33 Trimpley Close, City: Kiddy, Phone: 997331
```

Note that `upper_bound` returns an iterator to the position *past* the last element `e`. We could have used `find` in place of `lower_bound` in the `for` loop, since find returns an iterator to the *first* element found, if it exists within the container.

3.2.8 Map

`map` stores elements via a `key:value` pairing. The key can be any data type, providing there exists a sorting criterion for it. The elements themselves do not need to have any such ordering since they are inserted and ordered depending on their key. So whereas with the previous examples regarding `set`s, we ordered elements via the `name` of each `Address`, because the key was the `Address` object itself, now we can store the `Address` objects via some independent key. For example, we could provide an identification number for each `Address` inserted; or come up with some other way of identifying a key for each element. As with `set`, `map` does not allow for duplicate elements; you have to use `multimap` for this. You use `map` by including `<map>`.

Since you add a key and a value, elements are added as a `pair` object. Let's begin with an example where we insert `Address` objects into a `map`, giving them an `int` key based on the order that they were inserted:

```
/* map1.cc
   Compiled using g++ map1.cc Address.cc -o map1 */
#include <map>
#include "AddressRepository.hh"

int main()
{
  std::map<int, Address> map1;
  /* first way of inserting an element: */
  map1.insert(std::pair<int, Address>(1, addr1));
  /* second way of inserting an element: */
  map1.insert(std::map<int, Address>::value_type(2, addr2));
  /* third way of inserting an element: */
  map1.insert(std::make_pair(3, addr3));
  /* finally, add the rest using the first method of insertion: */
  map1.insert(std::pair<int, Address>(4, addr4));
  map1.insert(std::pair<int, Address>(5, addr5));

  std::map<int, Address>::iterator pos;
  for (pos = map1.begin(); pos != map1.end(); ++pos)
    {
      cout << "(" << pos->first << ") ";
      Address addr = pos->second;
      addr.print();
    }
  exit(0);
}
```

Example 3.14: *map1.cc*

As you can see, there are a few more complexities to deal with with `map` regarding inserting elements. What exactly are we putting in the `map`? With the previous containers, we inserted the object in question - for containers of integers, we used integers; for containers of `Address` objects we added the `Address` object itself. However, since a `map` contains a `key:value` pairing, the most obvious way to store elements is in a *pair*; this is achieved by using the `pair` class. There are three ways to insert elements into a `map` (and `multimap`) in a `pair`. Let's look at them each in turn:

`pair` is defined in `<utility>`, although we don't need to include the header file since `map` and `multimap` include it. `pair` is used to store (not surprisingly) two values. The `insert` method

for `map` is the same as for the other containers - in other words, it takes one value, namely an element. However, since `map` has a `key:value` paring, we need some way to deal with this. In this instance, when we declare

```
map1.insert(std::pair<int, Address>(1, addr1));
```

we're saying that the object inserted is a `pair` object; it takes an `int` and and `Address` (in the declaration `<int, Address>`), the first element is `1` (the key) and the second element is `addr1` (the value).

Another way to insert a `pair` into a `map` is to use `value_type`, which is defined differently for different containers. For `map` and `multimap`, it is a `pair`. So the declaration `std::map<int, Address>::value_type(2, addr2)` in the second call to insert is saying that the arguments `2` and `addr2` are passed to the `map` container, which is implicitly a `pair` object per element inserted.

The third way of inserting a `pair` is to call `make_pair`, which simply returns a `pair` object from the two arguments passed to it.

So when we put objects into a `map`, we've got to put - yes, you've got it - a `pair` object. That's fine; so how do we access elements of a `pair`?

This is demonstrated in the `for` loop:

```
std::map<int, Address>::iterator pos;
for (pos = map1.begin(); pos != map1.end(); ++pos)
  {
    cout << "(" << pos->first << ") ";
    Address addr = pos->second;
    addr.print();
  }
```

We first create an `iterator` to be able to traverse the container, and in the body of the `for` loop make the usual calls to `begin()` and `end()`. The `key:value` pairings are stored in the members `first` (for the key) and `second` (for the value) of the `pair`.

The output is obvious, so it's been omitted here; it's just the different `Address` objects preceeded by `(n)` where `n` is the number assigned to the map for that specific element. The `Address` object with the name of `"Jane"` exists twice within the `map`; this is because we are now inserting elements according to a different sorting criterion - `int` - and since there are no duplicates (our integers go from 1 to 5 in the above example), all elements are inserted successfully. However, if we'd have decided to give *each* element a key value of 1 (or any other number, providing the number's the same), only one `Address` object would have been inserted.

Let's talk about efficiency. If elements are stored as `key:value` pairings, it makes sense that we'll be able to access keys efficiently; this is true, and can be achieved in logarithmic time. This raises an important issue: if we want to change the key, how will elements be reordered and be consistent?

The answer is that we *cannot* change the key. Instead we have to remove the element with the key we wish to change and insert a new key (the one we wish to change the old key to) with the old value.

3.2.9 Multimap

As `multiset` is similar to `set` except that it allows duplicate elements, `multimap` is similar to `map` except that it allows for duplicate keys. Like `multiset` (Section 3.2.7 [Multiset], page 48), accessing ranges of `pair` objects is done by using `lower_bound` and `upper_bound`. The search operations available for `set` are also available to `multimap`, so we'll not look at them here because the interface is exactly the same. Just be aware that since you'd do a search on (for

example) `lower_bound(key)`, that you'll have to get the `key` from the `pair` object representing the `multimaps` element using the `first` member of `pair`. Like `map`, you use a multimap in your code by including `<map>`.

3.3 Generic Algorithms and Function Objects

We'll deal with two major topics in one fell swoop in this section: generic algorithms and function objects. Although independent of each other, they work together very nicely and as a result they've been glued into the same section.

To begin with, we'll take a look at a few very simple examples of some arbitrary function objects, as well as explain the basic philosophy behind them. We'll then take a look at a few key generic algorithms, and then combine the two concepts together, using generic algorithms in unison with function objects.

3.3.1 Function Objects - in a Nutshell

If you are familiar with the concept of a function pointer, then function objects aren't too disimilar. The basic concept holds for both: we can create many different instances of the same function, but each operating with a different state. However, the difference is that a function object is *not* a function - it is an object that *behaves* like a function.

We'll begin by describing some predefined function objects and look at how they work, and afterwards look at how to create our own function objects. This will all tie in with the next section, Section 3.3.4 [Introducing Generic Algorithms], page 57, which will show you how to exploit some of the function objects we'll look at in this chapter - many generic algorithms accept a function object as one of their arguments.

The STL provides a number predefined function objects. Our purpose here is to describe a few of them; the rest will be provided in the Section 3.3.2 [Some Predefined Function Objects], page 55.

Right. So we said that a function object is a class that behaves *like* a function. But it isn't *actually* a function; we're just wrapping up functional behaviour in a class by overloading the function call `operator()` so that we can call the class like we'd call a function. Let's look at a commonly used function, `negate`.

`negate` returns the negative value of what was passed in. Pretty simple really; let's look at the declaration in '`stl_function.h`' (don't despair at it - we'll explain it in due course!)[5]:

```
template <class T>
struct negate : public unary_function<T, T>
{
  T operator()(const T& x) const { return -x; }
};
```

Note that `negate` inherits from `unary_function` - in other words, it takes one parameter. Likewise, a binary function (which we'll meet soon) takes two parameters. `negate` deals with one type, `T`, the type we'll be passing in. This could be anything; an `int`, `string`, an `Address` object, and so on, with the following restriction: we must be able to negate the value of what we pass in (so for example we'd expect to get -5 if we passed in 5 as an `int`). We use the overloaded function operator `operator()` to be able to call `negate` as a function. Obviously, when we pass in `x`, we return `-x`.

[5] The declaration of `negate` has been modified to make it a little more readable; the exact text of `negate` will vary from implementation to implementation.

Let's see `negate` in use with a vector of integers, defining our own method named `with_each` to be able to walk through the elements of the container:

```
/* function1.cc
 * Compiled using g++ function1.cc -o function1 */
#include <vector>
#include <functional>

/* Define our own method that uses a unary function 'fn' on
 * a range of elements: */
template <class InputIterator, class Function>
void with_each(InputIterator beg, InputIterator end, Function fn)
{
  for( ; beg != end; ++beg)
    cout << fn(*beg) << endl;
}
int main()
{
  std::vector<int> v;
  v.push_back(10);
  v.push_back(2);
  v.push_back(4);
  with_each(v.begin(), v.end(), negate<int>());
  return 0;
}
```

Example 3.15: *function1.cc*

Here's the output:

```
$ ./function1
-10
-2
-4
```

The key to understanding what is going on involves looking at the `with_each` algorithm. Notice to start with that the third parameter of `with_each` in `main` takes `negate<int>()` as it's argument. This creates an instance of the `negate` function. The body of `with_each` works as follows:

```
for( ; beg != end; ++beg)
  cout << fn(*beg) << endl;
```

Here, `fn` is obviously `negate`, operating on the range of elements `beg` to (but not including[6]), `end`. It takes `*beg` as it's argument, which is a pointer to an integer. So all we're doing is saying "apply `fn` with argument `*beg`" - or, more precisely - "apply `negate` with argument `*beg`". More generally, `with_each` takes a function object, `fn`, so we could pass any unary function object into `with_each`.

`negate` is a unary function object. What about binary function objects? Well, there's not that much difference except that they take two arguments instead of one, surprise surprise. Let's look at a function object that takes two arguments, called `greater`. `greater` takes elements `x` and `y` and returns true if `x` is greater than `y`, false otherwise. It's definition is simple, and isn't too disimilar to `negate`:

[6] Recall from Section 3.2.2 [A Crash Course in Iterators], page 35 that `end` points past the *last* element in a container.

```
template <class T>
struct greater : public binary_function<T, T, bool>
{
  bool operator()(const T& x, const T& y) const
                { return x > y;}
};
```

We're going to hook `greater` into an example similar from earlier, using `greater` as an argument to a modified `with_each` method:

```
/* function2.cc
 * Compiled using g++ function2.cc -o function2 */
#include <vector>
#include <functional>

/* Define our own method that uses a binary function 'fn' on
 * a range of elements: */
template <class InputIterator, class T, class Function, class Message>
void with_each(InputIterator beg, InputIterator end,
       T val, Function fn, Message msg)
{
  for( ; beg != end; ++beg)
    if (fn(*beg, val))
      cout << *beg << msg << val << endl;
}

int main()
{
  std::vector<int> v;
  v.push_back(10);
  v.push_back(2);
  v.push_back(14);
  with_each(v.begin(), v.end(), 7, greater<int>(), " is greater than ");
  return 0;
}
```

Example 3.16: *function2.cc*

Again, the output is fairly obvious:

```
$ ./function2
10 is greater than 7
14 is greater than 7
```

... and isn't really anything to get excited about. We could have even passed in the predefined `less` function object, with `Message` being " is less than " and the results would again have been obvious. But the point is that function objects provide great flexibility and power, more so when we introduce generic algorithms.

This has been a whistle-stop tour to look describe some function objects and look at the basic principles involved. The idea is simple - create a class that does what you want it to do, placing the functionality in the `operator()` body. It doesn't need to be a template class - although it's always advantageous if you can do so. If you're perplexed about why we used the `with_each` method with the function objects, don't worry because we're actually mimicing a generic algoritm called `for_each`, which we'll encounter soon - you'll see (in Section 3.3.4 [Introducing

Generic Algorithms], page 57) how we'll use function objects with generic algorithms in a similar manner to how we just used `with_each`.

This may not have seemed much of an in-depth or informative explanation of function objects, but we're really holding back until we look at generic algorithms. All you need to remember is that function objects wrap up functional behaviour in a class, and we can make calls to that class by calling the overloaded operator.

3.3.2 Some Predefined Function Objects

You might actually be wondering *why* we should even bother using function objects, since the examples so far have been fairly simple and non-informative. Well, you can use them with most of the containers we have already seen. In Section 3.5.1 [Container Summary], page 68, one of the constructors for `set` and `map` passed in a *comparison* object; `list` provides it's own `sort` method, which you could pass a comparrison object to tell it how to sort data. Well, we can use any of the comparison function objects given here; so, instead of ordering using less-than (the default ordering), we can instead use `greater`, for example:

```
/* Create a set in which we sort elements using 'greater'
 * rather than 'less than': */
std::set<int, greater<int>()> some_set;

/* Now create a list: */
std::list<int> some_list;
/* ... add some elements to the list... */

/* Sort elements from greatest to least: */
some_list.sort(greater<int>());
```

The above example demonstrates (without going into too much detail) how the `greater` function object can be used to change the sorting criterion for a set, as well as sorting a list using `greater`. `greater`, along with a number of other predefined function objects, are detailed in Section 3.5.2 [Function Object Summary], page 71.

However, there are also a few other function objects that are worth metioning here, and are very useful. These are covered in the next section.

3.3.3 Function Adaptors

Function adpators are function objects that enable us to pass in other function objects as arguments, and there are a few worth mentioning that are extremely useful. The first set enable us to *bind* different arguments passed in (`bind1st` and `bind2nd`) to operations; the second set let us pass member functions as arguments (`mem_fun` and `mem_fun_ref`). Let's look at them each in turn.

Recall from Section 3.3.1 [Function Objects - in a Nutshell], page 52 in '`function2.cc`' we defined our own `with_each` method that took an additional parameter of type `T` so that we could pass in the value to make a comparison against the elements of the container, using the `greater` function object. The reason we did this was because `greater` is a binary function object, and it needs a second value to compare against, and so we provided it by supplying it a parameter. But the only way of doing this would be to modify the `with_each` method to cope with the extra parameter, which is what we did. But wait - look what happens if we change `with_each` as follows:

```
/* function3.cc
 * Compiled using g++ function3.cc -o function3 */
```

```
#include <vector>
#include <functional>

/* Define our own method that uses a unary function 'fn' on
 * a range of elements: */
template <class InputIterator, class UnaryFn, class Message>
void with_each(InputIterator beg, InputIterator end, UnaryFn fn,
       Message msg)
{
  for( ; beg != end; ++beg)
    if (fn(*beg))
      cout << msg << *beg << endl;
}

int main()
{
  std::vector<int> v;
  v.push_back(10);
  v.push_back(2);
  v.push_back(14);
  with_each(v.begin(), v.end(),
    bind1st(greater<int>(), 7), "7 is greater than ");
  return 0;
}
```

Example 3.17: *function3.cc*

This may seem esoteric, but what's actually happening is `bind1st` is transforming `greater` into a unary function object. The effect of `bind1st(op, val)` is to turn `op` into a unary function object such that `op` will work with the parameters `op(val, param)`. Thus, `val` will take on the value 7, and `param` will be whatever value we're working with inside the body of the `with_each` method, in other words the value held in `*beg`. Thus, when `fn(*beg)` is called in the body of `with_each`, we're calling `greater` with one argument (because `bind1st` turned it into a unary function), and the condition `if (fn(*beg))` yields true for all values that are greater than 7.

`bind2nd` is similar except that it binds its *second* parameter to be the *first* argument to be used in function `fn`. In other words, it transforms `bind2nd(op, val)` into `op(param, val)`.

Lets now look at `mem_fun_ref` and `mem_fun`.

The easiest way to describe `mem_fun_ref` is to revisit the `Address` class. First, recall from Section 3.2.3 [Vector], page 37 how we printed `Address` objects:

```
/* Declare an iterator to work with: */
std::vector<Address>::iterator pos;
/* Loop through the vector printing out elements: */
cout << "First iteration" << endl;
for (pos=v.begin(); pos<v.end(); ++pos)
  {
    pos->print();
  }
```

There's an easier way to do this, using `mem_fun_ref`. In the following code, we're actually going to use `with_each` again, which may seem counter-intuitive; well, it is if every time we're going to make function object calls to iterate a collection of objects we have to define code to

do so. `with_each` is just a stub for a generic algorithm called `for_each`, which we'll be looking at soon. Let's pass `print` to `mem_fun_ref` as we traverse through a vector of `Address` objects:

```
/* function4.cc
   Compiled using g++ function4.cc Address.cc -o function4 */
#include <functional>
#include <vector>
#include "AddressRepository.hh"

/* Define our own method that uses a unary function 'fn' on
 * a range of elements: */
template <class InputIterator, class UnaryFn>
void with_each(InputIterator beg, InputIterator end, UnaryFn fn)
{
  for( ; beg != end; ++beg)
    fn(*beg);
}
int main()
{
  vector<Address> v;
  /* Add all of the address objects to the vector: */
  v.push_back(addr1);
  v.push_back(addr2);
  v.push_back(addr3);
  v.push_back(addr4);
  v.push_back(addr5);
  /* Now call 'print' with each element, passing it by reference: */
  with_each(v.begin(), v.end(), mem_fun_ref(&Address::print));
  exit(0);
}
```

Example 3.18: *function4.cc*

The output is obvious: it prints out the list of `Address` objects. This is an extremely useful function object, because otherwise we'd have to define our own function object (called `fun_ob_print_address` within the `Address` class for example) which would do this work for us with the `with_each` algorithm, if we weren't happy with the `pos->print()` way of doing things. This adds extra code and is uneccessary if we can use `mem_fun_ref`.

`mem_fun` isn't too different; but instead of using a reference, `mem_fun` uses a pointer to an element.

Note that with both `mem_fun` and `mem_fun_ref`, the called member functions must be constant member function, otherwise a compile error will result.

These function adaptors are summarised in Section 3.5.2.2 [Function Adaptor Reference], page 72.

3.3.4 Introducing Generic Algorithms

Although the containers available provide a number of different useful methods, there are times when we need something a little more, when the container does not provide the necessary facilities to perform some operation. As you'll see, generic algorithms provide a strong set of tools to work with.

In case you are wondering, *generic algorithms* are exactly what they say they are; algorithms to provide some form of computation, available (under the right circumstances) to many types of data. Thus, a generic algorithm to sort a set of data should be able to compute on a character array; on an array of integers; on some container of objects, and so on.

In fact, we don't actually use data structures directly with generic algorithms. All algorithms accept (at the least) iterators as their arguments, and the type of iterator used with the algorithm determines what containers (and more generally, any object that uses iterators) can be used with the generic algorithm.

For example, consider `list`, which uses bidirectional iterators. Can we use the `sort` algorithm with `list`? Let's check the interface to `sort`:

```
template <typename RandomAccessIterator>
void sort(RandomAccessIterator first,
          RandomAccessIterator last);
```

Since the interface declares that we need `RandomAccessIterator`, using the `sort` algorithm with `list` is impossible - a compile error will result because `list` uses bidirectional iterators. This means that we can only use `sort` with `vector` and `deque`. This doesn't mean that we can't sort a list - in fact, `list` provides it's own `sort` method. If you're wondering how to sort `set` and `map` (because both use bidirectional iterators like `list`), remember that elements are inserted according to their value anyway, so are sorted automatically using a criterion defined by yourself.

There are many algorithms available to us, far too many to describe here. So instead we'll look closely at a few key algorithms, providing plenty of examples, and provide a summary of the rest of the algorithms in Section 3.5.3 [Generic Algorithm Summary], page 72, should you wish to explore them.

We'll look at the following generic algorithms here just to get a taste of them:

- Section 3.3.5 [for_each], page 58 enables us to walk through collections of objects, enabling us to perfrom some form of computation on each element of the collection.
- Section 3.3.6 [find], page 59 searches for some specified element.
- Section 3.3.7 [transform], page 61 let's us take some range, and copies the result of calling some operation `op` to another destination (in fact, both ranges can be the same).
- Section 3.3.8 [partition], page 62 enables us to move through a range of elements, moving elements that satisfy some unary predicate to the start of the range, leaving the rest of the elements to sit at the end of the range.
- Section 3.3.9 [accumulate], page 63 is a numerical algorithm that we can use to move through a range and *accumulate* some sum as we move through it.

3.3.5 for_each

The `for_each` algorithm, as you would expect, marches through some range performing some operation on each element as it goes. Sounds vague? That's because we can choose to do what we want *for each* element. We could march through a range printing the elements out; or modify the contents of each element.

In fact, `for_each` may sound familiar. It should be! We used a function earlier called `with_each`, in Section 3.3.1 [Function Objects - in a Nutshell], page 52. This may appear a bit cheaky, but we didn't actually *know* about generic algorithms when we were discussing function objects, and instead provided a quick-stub solution.

Let's look at the signature of `for_each` first:

`UnaryProc` **for_each**`(InputIterator beg, InputIterator end, UnaryProc op)`

As you can see, `for_each` takes two `InputIterator`s, meaning that we can use any container or object that uses `InputIterators`. `op` is the key point to the algorithm. We use `op`, a unary function to operate on each element; it is a unary operator because it takes one argument; namely, the element under scrutiny - in almost all respects, it's exactly the same as `with_each`, with the difference that `for_each` is already defined for us. There is also a `for_each` algorithm that accepts a binary operation `op`. Let's look at a simple example.

```
/* generic1.cc
 * Compiled using gcc generic1.cc Address.cc -o generic1 */
#include <algorithm>
#include <vector>
#include "AddressRepository.hh"

int main()
{
  std::vector<Address> v;
  v.push_back(addr1);
  v.push_back(addr2);
  v.push_back(addr3);
  v.push_back(addr4);
  for_each(v.begin(), v.end(), mem_fun_ref(&Address::print));
}
```

Example 3.19: *generic1.cc - printing out elements using* `for_each`

It's pretty straightforward really - and looks strikingly similar to '`function4.cc`' from Section 3.3.3 [Function Adaptors], page 55, except that we don't need to define our own method to march through a range of elements because `for_each` does it for us (notice we utilise `mem_fun_ref` to pass the member function `print` to `for_each`). Comparing this with the earlier method of using `pos->print()` and using `with_each`, you'll agree that it's easy on the eyes if nothing else.

3.3.6 find

Now that you've had a brief taster with `for_each`, everything get's a little easier. Understanding the signature of different algorithms becomes instinctively easier the more you use them. `find` has the following signature:

`InputIterator` **find**`(InputIterator begin, InputIterator end, const T& val)`

It's very straight forward: supply some start and end range, a value to search for, and return the position of where the element was, (`find` returns `end()` if the element was not found). There is also another algorithm called `find_if` that uses some operation to perform a test on the element being searched for:

`InputIterator` **find_if**`(InputIterator begin, InputIterator end, UnaryPredicate op)`

Let's look at `find`:

```
/* generic2.cc
 * Compiled using g++ generic2.cc Address.cc -o generic2
 * Run using ./generic2 */
#include <algorithm>
#include <list>
#include "AddressRepository.hh"

int main()
```

```
{
  list<Address> l;
  l.push_front(addr1);
  l.push_front(addr2);
  l.push_front(addr3);
  l.push_front(addr4);
  l.push_front(addr5);
  Address addr6("Jane", "55 Almond Terrace", "Worcs", 242783);
  list<Address>::iterator pos = find(l.begin(), l.end(), addr6);
  if (pos != l.end())
    {
      cout << "Found : ";
      pos->print();
    }
  return 0;
}
```

Example 3.20: *generic2.cc*

Once again, the example is easy to follow: we just supply the populated list to the `find` algorithm using `begin()` and `end()` and specify that we want to find a match with `addr6`. Note that since we test only for the name of an `Address` object, *any* `Address` object with the name "Jane" would have satisfied the search.

What about `find_if`? It's pretty straight forward really, and involves using `bind2nd()`:

```
/* generic3.cc
 * Compiled using g++ generic3.cc Address.cc -o generic3
 * Run using ./generic3 */
#include <algorithm>
#include <functional>
#include <list>
#include "AddressRepository.hh"

int main()
{
  list<Address> l;
  l.push_front(addr1);
  l.push_front(addr2);
  l.push_front(addr3);
  l.push_front(addr4);
  l.push_front(addr5);
  Address addr6("Jane", "55 Almond Terrace", "Worcs", 242783);
  list<Address>::iterator pos = find_if(l.begin(), l.end(),
bind2nd(less<Address>(), addr6));
  if (pos != l.end())
    {
      cout << "Found : ";
      pos->print();
    }
  return 0;
}
```

Example 3.21: *generic3.cc*

The code is almost the same as the previous example using `find()`, except this time we need a unary predicate as the third argument to `find_if()`. We achieve this by saying that we want to find any `Address` object less than `addr6`, in other words any `Address` object less than `"Jane"`. Since `"Bob"` is the first element we'll encounter and is less than `"Jane"` according to our less-than sorting criterion, `pos` is assigned to the iterator that points to `"Bob"`.

3.3.7 transform

`transform` isn't too disimilar to `for_each`, in the sense that both let us modify some range of elements. However, whereas with `for_each` we are modifying the range we pass in, with `transform` we can make changes to another range of elements. The signature for passing in a unary operation is as follows:

```
OutputIterator transform(InputIterator begin,
                         InputIterator end,
                         OutputIterator result,
                         UnaryOp op);
```

...which applies `op` for each element in the range `begin` to `end`, writing the result of each application of `op` to `result`. There is also a version of `transform` that takes a binary operation:

```
OutputIterator transform(InputIterator1 begin1,
                         InputIterator1 end1,
                         InputIterator2 begin2,
                         OutputIterator result,
                         BinaryOp op);
```

...which, for all elements in the range `begin1` to `end1`, applies `op` on each element in the range with, starting with `begin1` and `begin2`, up to `end1`.

Here's a simple example:

```
/* generic4.cc
 * Compiled using g++ generic4.cc -o generic4
 * Run using ./generic4 */
#include <algorithm>
#include <functional>
#include <vector>

int main()
{
  std::vector<int> v1, v2(10, 0);
  for (int i=0; i<10; i++)
      v1.push_back(i*10);

  transform(v1.begin(), v1.end(), v2.begin(),
            bind2nd(divides<int>(), 10));
  std::vector<int>::iterator pos;
  for (pos = v2.begin(); pos != v2.end(); ++pos)
    cout << *pos << " ";
  return 0;
}
```

Example 3.22: *generic4.cc*

We declare two vectors, populating `v2` with 10 elements each with a value of 0. The first vector is initialised with the values 0 through 90, incrementing by ten for each element. We use

the `bind2nd` function adaptor to divide each value of `v1` that we encounter by 10, and the result is copied into `v2`. The end result is that `v2` contains copies of all the values held in `v1`, divided by 10:

```
0 1 2 3 4 5 6 7 8 9
```

3.3.8 partition

`partition` takes all elements in a range and moves those that satisfy some unary predicate `op` to the front of the range `begin`; all other elements sit at the back of the range. The criterion can be a function object that we can use to test the elements in the container. For example, we could use `less` with `bind2nd` and some value that we want to test against for a container of integers.

Here's the signature of `partition`:

```
BidirectionalIterator partition(BidirectionalIterator begin,
                                BidirectionalIterator end,
                                Operation op);
```

Here's a simple example that uses integers:

```
/* generic5.cc
 * Compiled using g++ generic5.cc -o generic5
 * Run using ./generic5 */

#include <vector>
#include <algorithm>
#include <functional>

int main()
{
  std::vector<int> v;
  for (int i=9; i>-1; i--)
    v.push_back(i);
  std::vector<int>::iterator pos, iter;
  cout << "v: ";
  for (pos = v.begin(); pos != v.end(); ++pos)
    cout << *pos << " ";
  pos = partition(v.begin(), v.end(), bind2nd(less<int>(), 5));
  cout << "\nv, after partition: ";
  for (iter = v.begin(); iter != v.end(); ++iter)
    cout << *iter << " ";
  cout << "\nFirst element not matching in v: " << *pos << endl;
  return 0;
}
```

Example 3.23: *generic5.cc*

The output may seem strange at first:

```
v: 9 8 7 6 5 4 3 2 1 0
v, after partition: 0 1 2 3 4 5 6 7 8 9
First element not matching in v: 5
```

- it appears more or less as if we've performed a sort on the elements of the vector; and in a sense we have. We have said that we want to partition the data into two discreet sections; that which is less than five, and that which is greater than four. However, `partition` also sorts the

elements, and as a result the ordering of the original elements in the partition is not preserved, unlike with `stable_partition`, which is similar (and the signature is the same as `partition`), but preserves the order of the elements in the partition:

```
/* generic6.cc
 * Compiled using g++ generic6.cc -o generic6
 * Run using ./generic6 */

#include <vector>
#include <algorithm>
#include <functional>

int main()
{
  std::vector<int> v;
  for (int i=9; i>-1; i--)
    v.push_back(i);
  std::vector<int>::iterator pos, iter;
  cout << "v: ";
  for (pos = v.begin(); pos != v.end(); ++pos)
    cout << *pos << " ";
  pos = stable_partition(v.begin(), v.end(), bind2nd(less<int>(), 5));
  cout << "\nv, after stable_partition: ";
  for (iter = v.begin(); iter != v.end(); ++iter)
    cout << *iter << " ";
  cout << "\nFirst element not matching in v: " << *pos << endl;
  return 0;
}
```

Example 3.24: *generic6.cc*

The result is a little more predictable:

```
v: 9 8 7 6 5 4 3 2 1 0
v, after stable_partition: 4 3 2 1 0 9 8 7 6 5
First element not matching in v: 9
```

As a result, the time complexity of `stable_partition` is better than that of `partition`; `stable_partition` has linear complexity (worst-case is `n log n`), whereas `partition` has linear time complexity, worst-case number of elements / 2 swaps.

3.3.9 accumulate

Simple numerical processing is provided by `accumulate`. There are two forms that we should be aware of. The first takes some initial value, and computes the sum of this value with the elements in the range that were passed in:

```
T accumulate(InputIterator beg,
             InputIterator end, T val);
```

The second form of accumulate accepts not only an initial value and a range, but also a binary function operator, so that we aren't restricted to just making a sum:

```
T accumulate(InputIterator beg,
             InputIterator end,
             T val, BinaryFunc op);
```

Both of these algorithms are illustrated below:

```
/* generic7.cc
 * Compiled using g++ generic7.cc -o generic7
 * Run using ./generic7 */

#include <vector>
#include <algorithm>
#include <functional>
#include <numeric>

int main()
{
  std::vector<int> v;
  for (int i=0; i<10; i++)
    v.push_back(i+1);
  cout << accumulate(v.begin(), v.end(), 0) << endl
       << accumulate(v.begin(), v.end(), 1, multiplies<int>());
  return 0;
}
```

Example 3.25: *generic7.cc*

The output is trivial, and involves printing out 55 (starting at 1 and adding each number through to 10) and 3628800 (1 through to 10, but instead of adding, we use multiplication). We've given 1 as the value to the second `accumulate` call because it's the identity operator for multiplication.

3.3.10 Other Generic Algorithms

`for_each`, `find`, `transform`, `partition` and `accumulate` are just a few of the generic algorithms provided. The signatures, along with brief descriptions of all the generic algorithms, are given in Section 3.5.3 [Generic Algorithm Summary], page 72.

3.4 Strings

Of all the classes available to us in the STL, perhaps the most immediately useful is `string`. Using strings is often a necessity in many programs, and breaking away from the traditional `char *` declarations and having to worry about memory allocation and freeing are some of the many pitfalls that can cause code to become buggy and unpredictable. The `string` class provides us with an easy-to-use interface, making string-handling much less complex. In addition, it enables us to perform many different operations that we have seen previously using iterators, function objects and generic algorithms. Because of this we can manipulate strings in a fairly complex way, without too much code.

There are two types of string available to us; `string` and `wstring`. `wstring` is the implementation of strings that use more than one byte per character, such as unicode characters. We'll only look at using `string` here.

3.4.1 Basic String Usage

Constructing a string object is easy, and the following code shows a number of different ways to create strings, and perform some simple operations on them:

```
/* string1.cc
   Compiled using g++ string1.cc -o string1 */
#include <string>

int main()
{
  char *cstring = "third=c_string\0";
  string first("first");
  string second("2nd string", 3);
  string third(cstring);
  string fourth(4, '4');
  string five("This is the fifth");
  string fifth(five, five.find("fifth"));

  cout << first << ", " << second << ", "
       << third << ", " << fourth << ", " << fifth << endl;

  string line(first+", "+second+", "+third.substr(0, 5)+
      ", "+fourth.substr(3)+"th, "+fifth);
  cout << line << endl;
}
```

Example 3.26: *string1.cc: examples of creating strings*

The output is as follows:

```
$ ./string1
first, 2nd, third=c_string, 4444, fifth
first, 2nd, third, 4th, fifth
```

We begin by creating a C string, then follow up by creating five `string` objects. The first string we create contains the character array we pass in - `"first"`. The constructor for `second` takes a character array and initialises the string to have the first 3 characters from the array, so it contains the string `"2nd"`. The constructor for `third` takes the C string `"third=c_string\0"`. The string variable `fourth` is created by passing a character and a number; the resultant string is made up the character repeated `n` times (so in this case, `fouth` is made up of `"4444"`. Finally, we create a string called `five` from the character array `"This is the fifth"`, and use that string to initialise the string `fifth`. Notice that we are making a call to `find` within the constructor to `fifth`; `find` returns the first position, if it exists, of the occurrence of the string passed in to it. Since the string `"fifth"` indeed exists, the result - 12 - is passed back, and `fifth` is created by taking the twelfth character (and all beyond) of string `five`.

We then print all of these initialised strings (see the output above), and create a new string called `line` that will contain copies of the original strings, slightly modified. We call `substr` twice within the creation of `line`; `substr`, when passed 2 integers, returns the string represented by the start of the first position, counting as many characters as are passed in for the second argument. So calling `substr(0, 5)` on `"third=c_string\0"` will return `"third"`. Just passing one integer to the call to `substr` means that we take all characters starting from the position passed in. So `substr(3)` on `"4444"` will return the element at index 3 and beyond, which is a '4'. The result is that `line`, when printed comes out as

```
first, 2nd, third, 4th, fifth
```

As you can see, we've performed some relatively complex string manipulations with just a few lines of code.

Let's look more closely now at finding items within a string. The previous example was contrived because we planned all along for things to go our way; by this, I mean that we *knew* that the `find` calls would return the values that we were interested in. But what value is returned when we fail to find a position within a string? The answer lies in looking at the value `npos`. `npos` is defined within the `string` namespace, and defines the maximum size a string can be. When a search function fails to find part of a string, it returns `npos`, which we need to check against in order to ascertain whether the find worked or not. At the surface level, it's very useful, although as we'll see in a minute, there are a few pitfalls to be wary of. First though, an example:

```
/* string2.cc
   Compiled using g++ string2.cc -o string2 */
#include <string>

int main()
{
  std::string::size_type i;
  string sentence("Mary had a little lamb, his\
 fleece was as white as snow...");
  i = sentence.find("You'll never find this...");

  if (i == std::string::npos)
    cout << "i == npos; failed to find string.\n";

  cout << sentence.substr(0, sentence.rfind(" lamb")) << " "
       << sentence.substr(sentence.find("fleece"), 6) << endl;

  cout << sentence.substr(0, sentence.rfind("Again, no such string"))
       << endl;

  i = 0;
  int num = 0;
  /* Find out how many 'a's there are in the string 'sentence': */
  while(i != std::string::npos)
    {
      i = sentence.find("a", i);
      if (i != std::string::npos)
{
  num++;
  i++;
}
    }
  cout << "Found " << num << " occurrences of 'a'" << endl;
  exit(0);
}
```

Example 3.27: *string2.cc: finding things within a string*

The output goes like this:

```
$ ./string2
i == npos; failed to find string.
Mary had a little fleece
Mary had a little lamb, his fleece was as white as snow...
```

```
Found 7 occurrences of 'a'
$
```

Let's discuss the code. To begin with we create a variable `i` of type `size_type` and assign it to `npos`. After declaring and initialising the string `sentence`, we run the `find` function on `sentence`, passing in the string `"You'll never find this..."`. The result is assigned to `i`. If the search would have succeeded, it would've returned the index of the first element of the string we're searching for; but since the string we're searching for does not exist within `sentence`, `npos` is returned, and the evaluation `i == std::string::npos` will be true, since `find` returns `npos` because it failed to find the search string.

The statement

```
cout << sentence.substr(0, sentence.rfind(" lamb")) << " "
     << sentence.substr(sentence.find("fleece"), 6) << endl;
```

includes a new function call to `rfind`, as well as doing some more substring manipulation. `rfind` is similar to find, except that it searches *back* through the string in question instead of forwards. It returns the first position of the string it is searching for as it occurs from the *end* of the string. Since the string `" lamb"` exists, `find` returns the relevant position and `"Mary had a little"` is retrieved as the substring. The second part of the `cout` statement uses the `find` method as we'd expect it to work, and `"fleece"` is extracted (recall that substring returns the string starting from the first argument and counting as many characters as there are in the second argument). The result is that the string `"Mary had a little fleece"` is printed.

However, the following statement

```
cout << sentence.substr(0, sentence.rfind("Again, no such string"))
     << endl;
```

is different and deceiving; we use `rfind` to look for a string that clearly isn't in the string `sentence`. Since we're trying to create a substring from the start of `sentence`, to `sentence.rfind("Again, no such string")`, what will be printed out? The answer is that the entire `sentence` string will be printed, because `rfind` failed and as a result returned `npos`. And because `npos` returns the maximum (unsigned) value of its type, and the length of `sentence` is clearly less than that value, the `cout` statement just prints out the string in its entirety.

What we *should* have done is something like this:

```
long pos = sentence.rfind("Again, no such string");
/* If we've found what we're looking for, print the string out,
   or do whatever else we want: */
if (pos != npos)
  cout << sentence.substr(0, pos)
       << endl;
else /* The find failed, so do something else ... */
```

So, be warned! Always check the return value of a `find()` or `rfind()` method, to see if it is equal to `npos` or not. If it is equal to `npos` and you don't check for it, the example code above in '`string2.cc`' illustrates what could happen.

These are just a few of the operations we can perform with a string, and there are plenty of others that are all just as intuitive to use, such as `insert()`, `erase()` and `replace()`, amongst many others; since they are easy to understand they're in the Section 3.5.4 [String Summary], page 72, if you want to see the full range of operations you can use.

In the last section of '`string2.cc`', we counted the number of occurrences of the character 'a' that occur within `sentence`. It's fairly routine what we're trying to achieve here, so no code-breakdown is necessary. However, this section of code is undeserving; we can greatly reduce the amount of code for such a simple operation by using string iterators and a generic algorithm. . .

3.4.2 Iterators and Generic Algorithms

In the previous section, Section 3.4.1 [Basic String Usage], page 64, the last part of the code in '`string2.cc`' looked at counting the number of times the letter 'a' occurred within the string `sentence`. Intuitively, it doesn't look too harmless; it's more than a few lines long, but such is the price to pay for looking for the character we're interested in. However, there is no need for this block of code; it's wasteful and uses up *far* too much space. Why? Because there is a much easier way to do this if you recall that there are a number of algorithms available to us from the previous section; so why not use them? The algorithm of interest is `count`; the following example uses `count` to do *exactly* what the previous block of code did in '`string2.cc`':

```
cout << "Found " << count(sentence.begin(), sentence.end(), 'a')
     << " occurrences of 'a'" << endl;
```

`count` is from `<algorithm>`, of course, and seeing it at work here makes it suddenly obvious how generic algorithms can be very potent things to use. Remember however that we'd have to put `#include <algorithm>` at the start of the source file for it to be able to use `count`. We're making calls to `begin` and `end` to make use of the iterators provided by `string`. In one fell swoop, we've hacked down a larger piece of code into a composite call to `count`.

What kind of iterator are we using? In Section 3.2.2 [A Crash Course in Iterators], page 35, there are a number of different kinds of iterator discussed, and each kind gives us a clue as to how we can use them. `string` uses a random access iterator, in the same way that `vector` and `deque` use them. Like the container classes that use random-access iterators, invalidation occurs when the iterators are reallocated for some reason (such as deleting elements etc.), so be careful how you use them.

Need to reverse a string? Use the `reverse()` generic algorithm, given in Section 3.5.3 [Generic Algorithm Summary], page 72. You can combine function objects with these algorithms just like the examples in Section 3.3.3 [Function Adaptors], page 55 - you are only limited by the type of iterator that the algorithm takes as a parameter. Note that although there are a number of searching algorithms available, the `string` class already provides for these (see Section 3.5.4 [String Summary], page 72).

The exciting thing about this is that we're making use of iterators and generic algorithms to make code more readable and less dense. You'll find the large collection of predefined `string` operations more than enough in most situations, so for the most part you might not need to look at generic algorithms to do your work; we've mentioned it hear in passing so that you're aware of the fact that since strings use iterators (`begin()` and `end()` are already defined for you), we can use most generic algorithms with the `string` class.

3.5 STL Reference Section

This section gives an overview of all of the commands used throughout this chapter regarding the different components of the STL that we've encountered.

3.5.1 Container Summary

Here is the complete list of methods available for containers (where `<E>` denotes that the type of data contained within the container is of type `E`):

Constructors

`container<E> c`

Create a container with no elements

`container<E> c(n)`

Create a container with `n` elements

`container<E> c1(c2)`

Create a container which is a copy of container `c2`

`container<E> c(n, elem)`

Create a container with `n` elements with value `elem`

Size and capacity operations

`size_type` **size**`() const`

Returns the number of elements in this container

`size_type` **max_size**`() const`

Returns the maximum number of elements that this container may contain

`size_type` **capacity**`() const`

Returns how many elements this container can possess *without* reallocation

`bool` **empty**`() const`

Returns true if this container contains no elements

`void` **reserve**`(size_type num)`

Reserves internal memory for at least *num* elements

Special associative container operations

These operations are only available to `set`, `multiset`, `map` and `multimap`.

`size_type` **count**`(const T& val)`

Returns the number of elements equal to `val`. For `set` and `multiset`, `T` is the type of the elements; for `map` and `multimap`, it is the type of the key. This method has linear time complexity

`iterator` **find**`(const T& val)`

Returns the position of the first element with value `val`; if it's not found, `end()` is returned. This method has logarithmic time complexity

`iterator` **lower_bound**`(const T& val)`

Returns the first position where a copy of `val` would be inserted; if `val` is not found, `end` is returned. This method has logarithmic time complexity

`iterator` **upper_bound**`(const T& val)`

Returns the last position where `val` would get inserted. This method has logarithmic time complexity

`pair<iterator, iterator>` **equal_range**`(const T& val)`

Returns the first and last positions where `val` would get inserted. This method has logarithmic time complexity

`key_compare` **key_comp**`()`

Returns the comparison criteria

`value_compare` **value_comp**`()`

Returns the object used for comparison criteria

Assignment operations

`container&` **operator=**`(const container& c)`

All elements of the container are assigned the elements of `c`

`void` **assign**`(size_type num, const T& val)`

Replace all elements of the container with `num` copies of `val`; this method is only available to `vector`, `deque`, `list` and `string`

`void` **assign**`(InputIterator beg, InputIterator end)`

Replace all elements of the container with the elements in the range `beg` - `end`; this method is only available to `vector`, `deque`, `list` and `string`

void **swap**(container& c)

Swap the contents of this container with container `c`

void **swap**(container& c1, container& c2)

Swaps the elements of `c1` and `c2`

Element access operations

reference **at**(size_type index)

Returns the element at `index`; modifications to the container after using this method can invalidate the reference. This method is only available to `vector`, `deque`, `list` and `string`

reference **operator[]**(size_type index)

Return the element with index `index`; the first element is index 0. This method is only available to `vector`, `deque`, `list` and `string`

T& **operator[]**(const key_type& key)

Retuns the value of `key` in a map. Used for associative arrays with `map` and `multimap`

reference **front**()

Returns the first element. This method is only available to `vector`, `deque` and `list`

reference **back**() Returns the last element. This method is only available to

Insertion and deletion operations

iterator **insert**(const T& val)

Insert a copy of `val` into an associative `multiset` or `multimap`.

pair<iterator, bool> **insert**(const T& val)

Insert a copy of `val`

iterator **insert**(iterator pos, const T& val)

Inserts `val` at position `pos`

void **insert**(iterator pos, size_type num, const T& val)

Insert `num` copies of value `val` starting at position `pos`

void **insert**(InputIterator beg, InputIterator end

Insert copies of all elements in the range `beg` - `end` into a set, multiset, map or multimap.

void **void**(iterator pos, iterator beg, InputIterator end

Insert at position `pos` copies of all elements in the range `beg` - `end`. Only provided for vectors, deques, lists and strings.

void **push_front**(const T& val)

Inserts `val` as the first element. Only provided by lists and deques.

void **push_back**(const T& val)

Inserts `val` as the last element of the container.

void **remove**(const T& val)

Remove all elements with value `val`. Provided by lists.

size_type **erase**(const T& val)

Remove all elements with value `val` from a set, multiset, map or multimap; it returns how many elements were deleted from the container. For maps and multimaps, `T` must be the key

void **erase**(iterator pos)

Removes the element at position `pos`; only available to set, multiset, map and multimap

iterator **erase**(iterator pos)

Removes the element at position pos; only available to vectors, deques, lists and strings

void **erase**(iterator beg, iterator end)

Removes the elements in the range beg end; only available to set, multiset, map and multimap

iterator **erase**(iterator beg, iterator end)

Removes the elements in the range beg end; only available to vectors, deques, lists and strings

void **pop_front**()

Removes the first element of the container; only available to deques and lists

void **pop_back**()

Removes the last element of the container; provided by vectors, deques and lists

void **resize**(size_type num)

Changes the number of elements in this container to num. Only provided by vectors, deques, lists and strings

size_type **resize**(size_type num, T val)

? Only provided by vectors, deques, lists and strings

void **clear**()

Removes all elements from the container

List operations

Iterator methods

iterator begin()

Return an iterator to the first element of the container

iterator end()

Return an iterator to the point past the *last* element of the container

3.5.2 Function Object Summary

3.5.2.1 Standard Function Objects

Other than the function objects that we considered in Section 3.3.2 [Some Predefined Function Objects], page 55, STL provides a number of different predefined function objects. Since they're fairly intuitive to use (as you've seen in Section 3.3.1 [Function Objects - in a Nutshell], page 52), they do not require much explanation (names like multiplies and divide are self-explanatory).

Name	Description	Unary/binary	Operator
plus	Returns the sum of the two operands passed to it	binary	+
minus	Returns the sum of subtracting the second operand from the first	binary	−
multiplies	Returns the product of the two operands passed to it	binary	*
divides	Returns the result of dividing the second operand from the first	binary	/
modulus	Returns the result of applying the modulus operator to the two operands.	binary	%
negate	Returns the negated value of the operand passed to it	unary	−

`equal_to`	Returns true if the two parameters are equal	binary	==
`not_equal_to`	Returns true if the two parameters are not equal	binary	!=
`greater`	Returns true if the first parameter is greater than the second	binary	>
`less`	Returns true if the first parameter is less than the second	binary	<
`greater_equal`	Returns true if the first parameter is greater than or equal to the second	binary	>=
`less_equal`	Returns true if the first parameter is less than or equal to the second	binary	<=
`logical_and`	Returns the result of *and*-ing the two parameters passed to it	binary	&&
`logical_or`	Returns the result of *or*-ing the two parameters passed to it	binary	||
`logical_not`	Returns the result of *not*-ting the parameter passed in	binary	!

3.5.2.2 Function Adaptor Reference

This completes the adaptors given in Section 3.3.3 [Function Adaptors], page 55.

Expression	**Effect**
mem_fun(op)	calls `op` as a constant member function for an object
mem_fun_ref(op)	calls `op` as a constant member function for an object that has a pointer to it
bind1st(op, val)	takes a binary function object `op` and a value `val`, and returns a unary function object with the *first* argument of `op` bound to `val`. Thus `bind1st(op, val)` becomes `op(val, param)`.
bind2nd(op, val)	takes a binary function object `op` and a value `val`, and returns a unary function object with the *second* argument of `op` bound to `val`. Thus `bind2nd(op, val)` becomes `op(param, val)`.

3.5.3 Generic Algorithm Summary

This section lists all of the generic algorithms available from the STL.

- Sorry - you'll have to wait!!! (- Rich)

3.5.4 String Summary

The `string` class provides a number of useful member functions that make string manipulation much easier than standard C-like library calls.

- Sorry - you'll have to wait (again)!!! (- Rich)

3.6 Further Reading

This chapter introduced, at a non-complex level, the STL. However, one chapter on such a subject is not enough; you'll realise this if you chance upon any decent books about the STL.

As a consequence, we've had to forfeit quality of description, because the STL is a *very* complex subject area. Many important features and notes have necessarily been missed out due to limitations of space; the result is that we've not seen the whole picture, but a very limited viewa of the STL. The following references point to a number of different sources to which you should turn for more elaborate descriptions of the concepts we've been looking at. These references are a *must* if you wish to enrich your understanding of the STL.

The following documentation refers directly to GNU '`libstdc++`':

There are a number of books available for the STL:

STL *Tutorial and Reference Guide, Second Edition*

The C++ *Standard Library: a tutorial and reference, Musser, D., Gillmer, J., Atul, S., Addison-Wesley*

Generic Programming and the STL, Austern, M., Addison-Wesley

4 The GNU Compiler Collection

This chapter introduces the most useful and frequently used options for source file compilation using GCC - the GNU Compiler Collection. Using GCC you can compile C, Objective C, C++, Java and Fortran source files. We'll also look at the internal workings of `gcc` works at a relatively low-level such that you will be able to understand the entire compilation process with ease.

Section 4.1 [An Introduction to GCC], page 75 introduces GCC at a very high level; if you're a bit puzzled about how the compilation process slots together, or unsure of how GCC can handle many different languages, this is the place to start. We'll then look at many of the basic commands available to us for compilation of C source files in Section 4.2 [GCC Commands], page 78. Emphasis will also be placed on what actually happens during compilation, Section 4.3 [GCC Internals], page 86, and we'll look closely at the different steps and processes involved with using `gcc`. Compiling Objective C, C++, Java and Fortran source files is dealt with in Section 4.4 [Integrated Languages], page 88. We'll then look at using some of the compilation options with the M4 sources, providing a more pragmatic guide to the compilation steps we've already seen, in Section 4.5 [Pulling it Together], page 95. A list of books and links is provided in Section 5.8 [Further Reading], page 125. Finally, Section 4.7 [GCC Summary], page 102 rounds off everything we've dealt with, and mentions some of the tools that make management of compilation less stressful.

This chapter is not a definitive reference; it is only an introduction to well-used commands, and if you are already familiar with `gcc`, you will want to skip this section and read Chapter 5 [GNU Make], page 103.

4.1 An Introduction to GCC

Here we'll briefly outline the historical development of GCC, as well as practical aspects of getting gcc if you do not already have it. There is also a brief review of the broader picture of GCCs components, which glosses over the overall compilation process so that you'll have a high-level view of what will come later.

4.1.1 History of GCC

GCC originated during the middle of the eighties with the Free Software Foundation (FSF). It was developed and written originally as a one-man effort by Richard Stallman, founder of the FSF. The original version contained over 110 thousand lines of code, and it took Stallman a year to complete. The first beta release for GCC was in March 1987 - this was release 0.9; version 1 came out two months later. GCC began to develop rapidly with help coming from programmers interested on working with compilers. GCC continued to develop until in 1997 Cygnus EGCS began working on their version of the compiler collection.

This caused a split in the development in GCC; the Free Software Foundation development of GCC continued, as did the EGCS project, unfortunately both in different directions. Version 1 of the an EGCS compiler was released in December 1997, and support for the FSF compiler floundered.

In April 1999, the EGCS steering committee was appointed by the FSF as the official GCC maintainer. At that time GCC was renamed from the "GNU C Compiler" to the "GNU Compiler Collection" and received a new mission statement.

From this point the development became one branch again, maintained and overseen by one group. GCC 2.95 was released shortly afterwards, and in June 2001 version 3.0 was released. Prior to GCC 3.0, Objective-C, C++, Fortran and Chill were all integrated into the collection; version 3.0 (June 2001) saw the casting away of Chill and Java was integrated into the collection.

4.1.2 Where to get GCC

GCC can be downloaded from <http://gcc.gnu.org>; however, it comes as a package with all popular GNU Linux distributions, so if you are running GNU Linux then the chances are you'll already have (and be using) it. Distributions are available as tar balls in source or binary format - check the web-site for file sizes. If you are not sure that you have GCC or not, try

```
$ gcc --version
```

and you will either get a version number, for example:

```
$ gcc –version
3.0
```

or an error message:

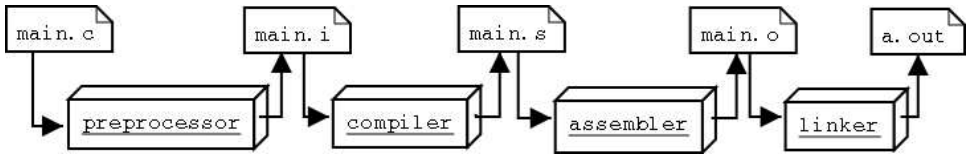
```
$ bash: gcc: command not found
```

4.1.3 A Brief Overview

How does GCC deal with compilation? How is it composed into smaller units (such as preprocessor, compiler etc.)? What is the basic design philosophy behind GCC? We'll review each of these questions briefly below.

4.1.3.1 The Broad Picture

Let's first of all focus on the larger picture of how GCC deals with compilation. The overall process of compilation involves a number of stages, shown in figure 1:



Example n: *Basic structure of GCC*

Figure 1 describes the compilation of a single C source file named '`main.c`'. It is first passed through the preprocessor, which pipes the result to the compiler, and a parse tree is built and the file is checked for syntactic and semantic errors. Providing the file is OK, RTL (register transfer language) code is produced from the parse tree and passed to the assembler, after the RTL has been turned into an assembler source file named '`main.s`' (it's a little more complicated than this, but the general picture holds). The assembler then translates the file into machine code, and '`main.o`' is produced. This is finally given to the link/load editor which produces the executable '`a.out`'.

Typically, we'd keep the source file, '`main.c`', and the binary, '`a.out`', and the temporary files listed above ('`main.i`', '`main.s`' and '`main.o`') would be piped from one stage to the next, or they'd be created as temporary files and removed when needed. However, there are plenty of options that enable us to stop at each phase of compilation and produce the relevant file for that stage.

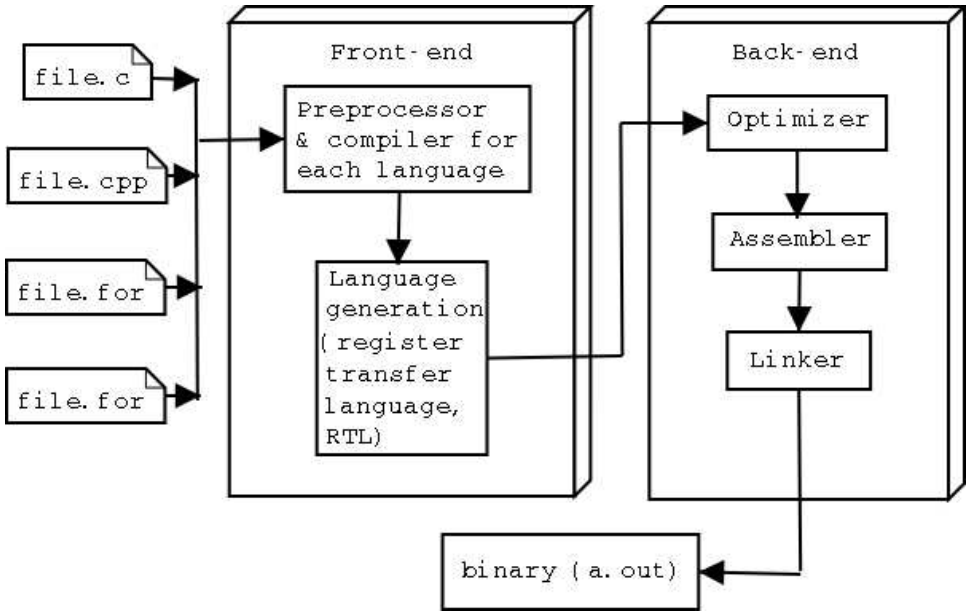
Although this example illustrates the compilation of a C source file, the same process holds forward for any of the languages used with GCC: take the source file and preprocess it if necessary[1]; send it to the compiler which then passes the file to the assembler, which in turn is

[1] Not all languages - such as Pascal - require preprocessing.

finally passed to the linker. An important need arises from this structure, to be able to make the compiler versatile - in the form of a *front* and *back* end of a compiler - which we'll now turn attention to.

4.1.3.2 Front and Back Ends

The phases of compilation warrant a further logical division; that of splitting the phases into a front and back end. The front end is concerned with the source language, and it's purpose is to preprocess the source file and derive a parse tree and to perform lexical, syntactic and semantic analysis on the code. This parse tree is then converted into the intermediate language, which is passed to the back end. The back end, on the other hand, is concerned with taking the (language-independent) code from the front-end, and preparing it in such a way that it can be converted into machine-dependant code. This is represented in diagram FIXME:



Example 1: *Front and back ends of a compiler*

Splitting it into two such sections is an efficient way of designing a compiler. Because the front end can pass on intermediate code to the back end, there is no need for any kind of language dependance on the part of the back end. All it sees is the intermediate language. You can also apply optimizations to this intermediate language. So what advantages do we gain from this? To begin with, applying optimization to the compilers language-independent code means that for any language you add to the compiler, all you have to do is parse it so that you can produce the intermediate code, and thus optimization does not need to be catered for specifically for each language. Also, it is easier to add new languages to the compiler: just write a suitable front-end that converts the source into some intermediate language (in the case of `gcc`, RTL), understood by the back end, and pass it on to the back end. The result is that you do not have to rewrite the compiler for each new language, because you have opted to use an intermediate language which saves you (greatly) in the long-run. For each platform you write a back end, all you need to know about is the intermediate language, and how to perform machine-based optimizations on it for the platform architecture.

This is the approach that GCC uses. When the parse tree is produced, RTL is produced as the tree is parsed. So regardless of whether we are compiling C, C++, Java, or any other source file that GCC is aware of, all the back end sees is RTL. The result is that we can provide optimizations

to RTL, thus only having to write the language-to-RTL front-end, and by providing a machine description for each architecture, the object code can easily be produced without much bother, from RTL. We simply apply optimizations on the RTL based on the architecture we're using. The end result is that for `j` languages and `k` architectures, we only need to put in `j+k` effort because of the benefits of performing optimization on the intermediate language, and each architecture will have a mapping from RTL to machine code[2].

The result is that GCC provdies a versatile collection of compilers for many different languages and architectures, neatly modularized and packaged. The next section will deal pragmatically with the GNU compiler collection, and emphasis will be placed on how the material here relates to GCC.

4.2 GCC Commands

This section looks at the practical use of `gcc`, as well as some of the more esoteric aspects of it's use. We'll first look at the different commands available with `gcc`, and will place emphasis on each of the different stages of compilation (Section 4.2 [GCC Commands], page 78). The internals of `gcc` are then covered (Section 4.3 [GCC Internals], page 86), and a discussion of the inner workings explains what happens when `gcc` is invoked. The different languages that are supported by `gcc` are also covered, and simple compilations are explained along with more commonly used options for each language (Section 4.4 [Integrated Languages], page 88).

`gcc` supports a plethora of command options; the `man` and `info` pages contain a full listing[3], and on-line documentation can be viewed at `http://gcc.gnu.org/onlinedocs`. Here, attention is drawn to the more useful and commonly used `gcc` commands, and compilation of C source files is assumed. Other languages are dealt with in Section 4.4 [Integrated Languages], page 88. More obtuse commands (for example platform options, code generation options etc.) are not dealt with here. You can find a summary of all the commands we'll look at in Section 4.6 [Reference Section], page 99.

4.2.1 Overview

Here we'll look indepth at each of the stages of compilation using `gcc`. It will be a very detailed and verbose study; so if you are already familiar with the different stages of compilation, the Section 4.6 [Reference Section], page 99 which contains all of the options discussed here, will be more appropriate.

However, if you have used a C compiler before, but have not had to deal with the different stages (for example, if you used an integrated development environment such as Borland's C compiler), or have only used an interpreted language such as Visual Basic, this section will give you all the detail necessary to use `gcc`s different compilation steps with a good degree of confidence.

4.2.2 Basic Compilation Options

`gcc` can be invoked in it's simplest form using the command:

```
gcc [options] source.c
```

where '`source.c`' is the name of the source file. `[options]` can be any of the options given in the succeeding sections. This command compiles the source file and produces an executable

[2] Contrast this with having to write an optimzier for each langauge for *each* architecture - for `j` languages and `k` architectures, we'll have to put in `j*k` effort.

[3] Type `man gcc` or `info gcc` for more information.

file named 'a.out' (due to historical reasons), providing there were no errors encountered in which case gcc will issue appropriate (or not as the case may be) error messages.

In this instance gcc preprocess the file and pipes it to the compiler; the compiler then produces a temporary file named 'source.s' containing the assembly code; this is then passed to the assembler and the object file 'source.o' is produced; finally, the executable 'a.out' is produced after linking and temporary files ('source.s' and 'source.o') are removed.

The following examples assume the compilation of one file. In practice many files would be included, in which case more versatile command line options may sometimes be required. Including multiple files is easy; just list the files one after another. Invoking

```
gcc [options] source1 [[source2] ... [sourcen]]
```

will compile 'source1' and optionally 'source2' up to '*sourcen*' to produce the executable 'a.out', providing no errors occurred and all external symbols were satisfied. Listing multiple files can be done for any of the commands discussed below with regard to the different stages of compilation. In practice with larger projects one would use a 'makefile' to manage multiple files - see Chapter 5 [GNU Make], page 103.

Although you may want to produce a binary most of the time, there are a number of options that enable you to stop at each different of the compilation. Each of these sections - preprocessing, compilation, assembly and linking - will be reviewed next, as well as some of the more well-used options.

4.2.3 The Preprocessor

Before any stages of compilation occur, you must first create a source file. This will be some high-level language ASCII text file containing your code. The purpose of the preprocessor is to perform textual substitutions within the code where needed. Textual substitution can come in a number of different forms: header file inclusion, macro substitution and comment substitution.

Produce preprocessed output by passing the -E flag to gcc. This will write to 'stdout' the contents of all source files after being preprocessed, stripping out comments. To produce preprocessed output with comments (such that the preprocessor does not strip them out), include the -C flag along with the -E option.

Although you can pass the -E flag to gcc, you can also invoke the preprocessor, 'cpp', directly. Try cpp --help for a full listing of options available for 'cpp'.

There are a number of reasons why you may want to view the output of the preprocessor. Firstly, it can reveal if any lines of code have been stripped out by the preprocessor due to bad commenting. Viewing preprocessed output can also be useful when viewing the results of expanding macros.

The preprocessed file includes a number of line numbers for any #include statements in the source file. For example, the output of the source file below:

```
/* main.c */
#include <stdio.h>
int main()
{
        return(0);
}
```

results in the following output for the preprocessed file when it's been filtered (using a utility like grep[4] for example) to search for all lines beginning with a '#':

[4] grep is used for searching patterns in a line

```
$ gcc -E main.c > main.i
$ grep '^#' main.i
# 1 "simple.c"
# 1 "/usr/include/stdio.h" 1 3
# 1 "/usr/include/features.h" 1 3
# 138 "/usr/include/features.h" 3
# 196 "/usr/include/features.h" 3
# 1 "/usr/include/sys/cdefs.h" 1 3
# 71 "/usr/include/sys/cdefs.h" 3
# 103 "/usr/include/sys/cdefs.h" 3
# 250 "/usr/include/features.h" 2 3
# 1 "/usr/include/gnu/stubs.h" 1 3
# 278 "/usr/include/features.h" 2 3
# 27 "/usr/include/stdio.h" 2 3
# 1 "/usr/lib/gcc-lib/i486-suse-linux/2.95.2/include/stddef.h" 1 3
# 19 "/usr/lib/gcc-lib/i486-suse-linux/2.95.2/include/stddef.h" 3
...
(the rest is left out because of space limitations.)
...
$
```

It is worth briefly mentioning what this means, since this convention is rather hermetic unless it has been encountered before. Such file inclusions are organised as follows:

 # line_number file_name flags

where the flags are defined as follows:

1 - The start of a new file

2 - Return to a new file after including another file

3 - The text included from this inclusion is a system header file; so supress certain warnings

4 - The following text should be treated as C

Thus, in the example above, `# 1 "/usr/include/stdio.h" 1 3` is saying "at line number 1 in '`stdio.h`', note the start of the file and also that this is a system header file"; note that '`stdio.h`' includes '`features.h`' (look at the flags and compare them with the descriptions above), which itself includes '`cdefs.h`', which returns to '`features.h`' and then includes '`stubs.h`', and finally returns to '`stdio.h`'.

Once the file has been preprocessed, and header files have been included and any macros have been expanded, it is passed on to the compiler.

4.2.4 The Compiler

The compiler's task is to produce assembly code from the preprocessed source file utilising several different stages: lexical, syntax and semantic analysis, intermediate code generation, optimization and code generation.

`gcc` uses an intermediate language known as RTL - register transfer language, during the compilation stage. RTL is produced as the parse tree is built, and as functions etc. are defined, they are turned into RTL instructions and a number of optimizations are performed on the RTL code.

To produce a file output from the compiler, invoke `gcc` using the `-S` flag. The output will be a number of files in assembler with the '`.s`' extension. You can pass either a '`.c`' or preprocessed '`.i`' file to the compiler; `gcc` will determine by the file extension what needs to be done in terms of whether or not to pass the file(s) to the preprocessor first or not. Producing assembler sources

may not seem very useful, but can be helpful if you want to write an assembler routine, and want to see how it is done (write a C program to do it, then run it through `gcc` with `-S` and check out the assembly language produced in the corresponding '`.s`' file).

As with the preprocessor, you don't need to pass the file(/s) directly to `gcc` - you can pass them to the compiler, '`cc1`', passing the `--help` flag to the compiler if you need to know the options available.

4.2.5 The Assembler

Since the output of the compilation process is a file (or possibly set of files) in assembly language for a specific machine, the penultimate stage before an executable program is produced is to take the assembler code and turn it into machine code. This is done for each source file passed to the compiler. At this stage the files are ignorant about any other files that they may be included with; possibly there will be a number of symbols that make references to other files. It is the purpose of the link/load editor to resolve these external references in the next stage, linking.

To produce the machine code file compile your program using the `-c` command. You can pass '`.c`', preprocessed '`.i`' or assembled '`.s`' files to the `gcc` using the `-c` flag; the output will be a file(s) with the extension '`.o`' containing object code. Once again, like the previous stages, you don't have to rely on `gcc` - you can invoke the assembler '`as`' directly (see Section 4.2.7 [Passing Arguments to the Assembler and Linker], page 82 on how to do this directly from `gcc`).

4.2.6 Link Editing And Libraries

The linking phase is the last phase before producing the executable file. The assembler will have taken the files in assembler and produced object files, the final stage involves linking all of these files together into the final binary.

If there is more than one object file passed to the linker or libraries have been linked, then there may be a number of external symbols to resolve. In fact, even if you pass one source file through `gcc`, there may be unresolved symbols if you are referencing functions or variables from libraries. An external sybmbol is simply a reference to a variable or function from one file to another file or library. The link/load editor attempts to resolve these references by searching the standard libraries and the object files that have been created as output from the assembler. If there are no unresolved symbols - in other words all references to external symbols were satisfied - an executable file is produced. Unresolved references will mean that the linker will inform you of the references that could not be resolved and no executable will be produced.

During linking, the linker will search all of the standard directories looking for specific libraries. You can include a library by using

`-lname`

When linking, use library '`libname.so`'. If '`libname.so`' cannot be found, use '`libname.a`'. By default, all libraries contain the prefix '`lib`' and the suffix '`.so`' or '`.a`'; for example `-lnsl` would look for '`libnsl.so`', the network service layer.

`gcc` will look in the standard directories looking for this library; if you want to specify a directory to be searched, use the flag

`-Ldir`

which will tell `gcc` to also look in directory '`dir`' when searching for libraries.

As mentioned previously, shared libraries will be linked first, unless none are found and static libraries are linked (if found). Use

`-static`

to indicate that static libraries should be linked *instead* of shared libraries (this option has no effect if the system does not support dynamic linking).

Like all the previous stages you can invoke the linker, '`ld`', directly, instead of relying on `gcc` (see Section 4.2.7 [Passing Arguments to the Assembler and Linker], page 82 on how to do this from `gcc`).

4.2.7 Passing Arguments to the Assembler and Linker

Although uncommon, you can also pass arguments directly from `gcc` to the assembler and linker.

To pass arguments to the assembler, use the command

```
-Wa,option-list
```

And to pass arguments to the linker, use the command

```
-Wl,option-list
```

where *option-list* is a list of comma separated options (with no white-space between options whatsoever) to be passed to either process.

A useful option to pass to the linker is when certain runtime shared libraries cannot be found; use the `-rpath PATH` to set this. For example,

```
$ gcc -Wl,-rpath /usr/lib/crti.o
```

tells `gcc` to pass the option `-rpath /usr/lib/crti.o` to the linker, such that the linker looks at '`/usr/lib/crti.o`' to be included in the runtime search path. Since there are a variety of useful options that you can pass to both assembler and linker, we'll not list any here and be satisfied with having explained that it's at least possible. To view a full listing of either assembler or linker, try

```
process --help
```

where *process* is one of '`as`' (the assembler) or '`ld`' (the linker).

4.2.8 Useful GCC Options

We've stepped through the commands necessary to do a basic compile and be able to pass source files through different stages of compilation; however, these commands on their own are not enough to be able to do anything very useful with. What follows is a list of common commands that you will find useful in passing source files through `gcc`.

4.2.8.1 C Language Features

Depending on how old your code is, and how much you have stayed in line with ANSI C, there are a number of switches to deal with compiling your C source files[5].

`-traditional` Supports the traditional C language, including lots of questionable, but common, practices. The traditional option also supports all of the FSF's extensions to the C language.

[5] The following options, `-traditional`, `-ansi` and `-pedantic`, are taken verbatim from the first edition. Also, see Section 2.2 [Standards Conformance], page 9 for a little bit more about the ISO standards etc.

`-ansi` Supports the ANSI C standard, though somewhat loosely. The FSF's extensions are recognised, except for a few that are incompatible with the ANSI standard. Thus ANSI programs compile correctly, but the compiler doesn't try too hard to reject non-conformatnt programs, or programs using non-ANSI features.

`-pedantic` Issues all the warning messages that are required by the ANSI C standard. Forbids the use of all the FSF extensions to the C language and considers the use of such extensions errors. It's arguable whether or not anyone wants this degree of conformity to the ANSI standard. The FSF obviously feels that it isn't really necessary; they write "This option is not intended to be *useful*; it exists only to satisfy pedants." We feel that it's useful to check for ANSI conformity at this level and also that it's useful to disable the FSF's own extensions to the language. As the `gcc` manual points out, `-pedantic` is not a complete check for ansi conformance - it only issues errors that are required by the ANSI standard.

4.2.8.2 Defining Constants

The `-D` option[6] acts like a `#define` in the source code, and can be used to set the value of a symbol on the command line:

`-DDEFN` Define `DEFN` to have the value 1; use with the preprocessor option `#if`.

`-DDEFN=VAL` Define `DEFN` to have the value `VAL`.

`-UDEFN` Undefine any constants with definition `DEFN` (all `-D` options are evaluated before any `-U` options on the command line).

For example, the command:

```
$ gcc -DDOC_FILE=\"info\" -DUSE_POLL file.c
```

sets `DOC_FILE` to the string "info" (the backslashes are there to make sure that they're interpreted part of the string). This can be useful for controlling which file a program opens, for example. The second `-D` option defines the `USE_POLL` sysmbol to have the value 1, and you use the `#if` directive to see whether `USE_POLL` (or any other value set by `-D`) is set.

4.2.8.3 Default File Renaming

To replace the name of any created default file (for example '`a.out`' from a full compilation, or '`source.s`' from using `-S` command with a file called '`source.c`', etc.), use the `-o` option:

```
gcc source.c -o prog
```

produces an executable file name '`prog`' after compiling the source file '`source.c`'. This replaces the default executable file named '`a.out`'.

Use the `-o` option in any of the stages which produce some output file to redirect the default naming conventions ('`.o`', '`.s`' etc.) to any file name specified by the `-o` flag. Thus,

```
gcc source.c -S -o newFile1
```

and

[6] This section is adapted from the first edition.

```
gcc source.c -c -o newFile2
```

will produce a file named '`newFile1`' and '`newFile2`' respectively instead of the default '`source.s`' and '`source.o`'.

4.2.8.4 Verbose Output

Use the `-v` option to print verbose information to the '`stderr`' output stream. This tells you (among other things) about:

- the version of `gcc` being used;
- the search directories (user and system) to look for header and other source files to be included during any of the stages of compilation
- the options specified (that are normally hidden fron the user) about the arguments passed to the C preprocessor, compiler, assembler and the linker.

4.2.8.5 Including Directories

`gcc` searches in the default directories for include files enclosed in `#include <...>` brackets (the `-v` flag details which directories are searched in case you do not know). For user defined headers enclosed in quotes, `gcc` looks in the directory of the current source file being scanned. Thus, the inclusion

```
#include "func.h"
```

will search in the current directory for a file named '`func.h`'. Simalarly,

```
#include "../../headers/func.h"
```

includes a file named '`func.h`' two levels up the directory tree in a directory named '`headers`' from the current directory of the source file declaring the inclusion. Since this introduces dependance on the way in which files are placed with regard to directory structure, it is better practice to use the `-I` flag which tells the compiler where to search. In the previous example, compiling with the options

```
gcc mainFile.c -I../../headers
```

tells the compiler to look in the '`../../headers`' directory for any files included by the programmer that are *not* found in the current directory. Specify the `-I` flag as many times as needed:

```
gcc mainFile.c -I../../headers -I../../defs -I../../gen
```

tells the compiler to search two directory levels up in the '`headers`', '`defs`' and '`gen`' directories.

Using `-I` will not search for any `#include <...>` files; use the `-I-` flag to tell the compiler that any `-I` commands following the `-I-` flag should also look for any `#include <...>`. files.

For example, the command

```
gcc main.c -I../../headers -I- -I../../defs -I../../gen
```

will seach for `#include "..."` files located in the '`../../headers`' directory, and search for `#include <...>` *and* `#include "..."` files located in the '`../../defs`' and '`../../gen`' directories. Note that the current directory will *not* be searched; to include the current directory, use `-I.`:

```
gcc main.c -I../../headers -I- -I../../defs -I../../gen -I.
```

4.2.8.6 Pipes

By default, temporary files are created during compilation and are removed at the end of compilation. Use the `-pipe` flag to pipe files through each stage rather than use temporary files.

4.2.8.7 Debug Information

`-g` tells `gcc` to provide debug information. Debuggging is covered in more detail in Chapter 14 [Debugging with gdb and DDD], page 247 and is therefore not part of this chapter.

4.2.8.8 Optimization

Chapter 15 [Profiling and Optimising Your Code], page 263 provides more information about optimization and is not covered in this chapter.

4.2.9 Warnings

Let's look at a few flags to control the level of warning we can control. Listed here are some of the more useful and commonly used warning options; if you want to view the full set of warnings available, the GCC '`man`' pages contain much more information.

All of the warning options are prefixed by `-W`. So (for example) `-Wall` literally stands for *all* warnings, and is not some euphemism for a wall!

`-w`	Inhibit all warning messages.
`-pedantic`	Print all warnings that are required by ANSI standard C.
`-pedantic-errors`	Like `-pedantic`, except that the warnings are instead treated as errors.
`-Wall`	This a very general warning option: it encompasses a whole range of features, each of which are listed below in following `-W...` options[7]:
`-Wimplicit-int`	Any function declarations without a return-type will be reported; their return type will default to `int`.
`-Wimplicit-function-declaration`	Warns whenever a function is encountered in a block of code, but has not yet been declared.
`-Wimplicit`	Same as `-Wimplicit-int -Wimplicit-function-declaration`.
`-Wmain`	Warns if `main`: - takes any other arguments other than (a) an `int` followed by `char **`, or (b) no arguments, or - returns any value other than `int`.
`-Wreturn-type`	This flag warns that any function not declaring a return-type defaults to `int`, and also warns of any non-void functions that do not return a value.
`-Wunused`	Warns if any: - local variable is declared but not used; - a static function is defined, but is not used; or - a statement computes a result, but the result is never used.

`-Wswitch`	Warns if - an enumeration is used in a switch statement, but at least one of the elements of the enumeration has not been used in the switch; or - some value not catered for by the enumeration is used in the same switch statement.
`-Wcomment`	Warns if any comment started with `/*` contains an inner `/*`; for example, `/* some comment... /* */`.
`-Wformat`	If any print-related functions are called (for example `printf`), type-mismatching of arguments is reported where it occurs.
`-Wchar-subscripts`	-
`-Wuninitialized`	An automatic variable is used without first being initialized.
`-Wparentheses`	Warns if parentheses are omitted in certain contexts.

4.3 GCC Internals

In practice, even a simple compilation produces half a page of text when ran with verbose output to explain what is happening at each stage of the compile. Thus, it is necessary to demystify this process and explain the inner workings of `gcc` when ran with a simple compilation. With this knowledge, you'll have a firmer understanding of how to examine and walk through detailed compilations, making (and tweaking) changes if need be.

Throughout this chapter the different parts of the compiler have been unveiled; first the preprocessor, then the compiler, and assembler, and so on. Each of these components is a spearate process; each has its own functionality and responsibility, which we'll now encounter with an example of a full compile with verbose output.

There are a number of processes to make note of when invoking `gcc` with a C source file; the preprocessor '`cpp`'; the C compiler, '`cc1`'; the GNU assembler, '`as`'; and the GNU linker, '`ld`'. Externally it is not necessary to know about these separate processes - `gcc` will take care of this without you needing to know. However, `gcc` supports various command line switches to enable the control of which parts of the compilation process are used. Although it may appear convenient to gloss over these details, knowing about them will make your awareness of how `gcc` works on a relativley low level much stronger, and if errors occur you'll be more prepared to deal with them.

Consider the following listing of a sample compilation:

```
$ gcc main.c -v
Reading specs from /usr/lib/gcc-lib/i486-suse-linux/2.95.2/specs
gcc version 2.95.2 19991024 (release)
 /usr/lib/gcc-lib/i486-suse-linux/2.95.2/cpp -lang-c -v -D__GNUC__=2
 -D__GNUC_MINOR__=95 -D__ELF__ -Dunix -D__i386__ -Dlinux -D__ELF__
 -D__unix__ -D__i386__ -D__linux__ -D__unix -D__linux -Asystem(posix)
 -Acpu(i386) -Amachine(i386) -Di386 -D__i386 -D__i386__ -Di486
 -D__i486 -D__i486__ main.c /tmp/ccxswtDi.i
GNU CPP version 2.95.2 19991024 (release) (i386 Linux/ELF)
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /usr/lib/gcc-lib/i486-suse-linux/2.95.2/include
```

```
 /usr/include
End of search list.
The following default directories have been omitted from the search
path:
/usr/include/g++
/usr/lib/gcc-lib/i486-suse-linux/2.95.2/../../../../
  i486-suse-linux/include
End of omitted list.
/usr/lib/gcc-lib/i486-suse-linux/2.95.2/cc1 /tmp/ccxswtDi.i -quiet
-dumpbase main.c -version -o /tmp/cc0pQ7AA.s
GNU C version 2.95.2 19991024 (release) (i486-suse-linux) compiled by
GNU C
version 2.95.2 19991024 (release).
/usr/i486-suse-linux/bin/as -V -Qy -o /tmp/ccwiNRyM.o /tmp/cc0pQ7AA.s
GNU assembler version 2.9.5 (i486-suse-linux) using BFD version
  2.9.5.0.24
/usr/lib/gcc-lib/i486-suse-linux/2.95.2/collect2 -m elf_i386
  -dynamic-linker
  /lib/ld-linux.so.2 /usr/lib/crt1.o /usr/lib/crti.o
  /usr/lib/gcc-lib/i486-suse-linux/2.95.2/crtbegin.o
  -L/usr/lib/gcc-lib/i486-suse-linux/2.95.2 -L/usr/i486-suse-linux/lib
  /tmp/ccwiNRyM.o -lgcc -lc -lgcc
  /usr/lib/gcc-lib/i486-suse-linux/2.95.2/crtend.o  /usr/lib/crtn.o
$
```

Don't be intimidated! Although the output may appear daunting at first, the process when broken into logical divisions becomes much easier to understand. Since we've already discussed the different phases of compilation, all we really need to do is pick out which sections of the compilation are dealing with specific phases. It couldn't be easier; looking at the compilation, there are a number of points to make:

- The '`specs`' file

  The first thing that is read is the '`specs`' file. It's purpose is to define rules for compilation. It is extremely unlikely that you'll need to tweak the '`specs`' file, and you should avoid it unless necessary. And if you do know how to tweak it, chances are you don't need to read this section...

- '`cpp`', the C preprocessor

  The section

  ```
  /usr/lib/gcc-lib/i486-suse-linux/2.95.2/cpp -lang-c -v -D__GNUC__=2
    -D__GNUC_MINOR__=95 -D__ELF__ -Dunix -D__i386__ -Dlinux -D__ELF__
    -D__unix__ -D__i386__ -D__linux__ -D__unix -D__linux
    -Asystem(posix) -Acpu(i386) -Amachine(i386) -Di386 -D__i386
    -D__i386__  -Di486 -D__i486 -D__i486__ main.c /tmp/ccxswtDi.i
  ```

  which calls the preprocessor, '`cpp`', with a host of preprocessor options, passing '`main.c`' through the preprocessor and creating a file named '`/tmp/ccxswtDi.i`'. Notice that CPP passes `-lang-c` as one of its options; it's saying "the following source file is a C file". Calling any of the integrated languages will ensure that the `-lang-...` command will be called, providing that `gcc` recognises the file extension for that source file.

  The preprocessed file is placed in a temporary file named '`/tmp/ccxswtDi.i`'; this will be used by the compiler to generate assembly code. Note here that each `-D` instance is defining a constant (see Section 4.2.8.2 [Defining Constants], page 83); there are also a number of

architectural flags (`-A`). There's no real need to have to worry about these flags - they are dealt with for you, and you shouldn't have to bother with them.

- '`cc1`', the C compiler

  With '`/tmp/ccxswtDi.i`' now created, it is passed to the C compiler; it's output is to produce a file named '`/tmp/cc0pQ7AA.s`' which will contain the assembly code:

  ```
  /usr/lib/gcc-lib/i486-suse-linux/2.95.2/cc1 /tmp/ccxswtDi.i -quiet
  -dumpbase main.c -version -o /tmp/cc0pQ7AA.s
  ```

  It creates as its output file '`/tmp/cc0pQ7AA.s`', the assembly file.

- '`as`', the GNU assembler

  '`/tmp/cc0pQ7AA.s`' is now passed to the assembler to produce an object file, named '`/tmp/ccwiNRyM.o`':

  ```
  /usr/i486-suse-linux/bin/as -V -Qy -o /tmp/ccwiNRyM.o
    /tmp/cc0pQ7AA.s
  ```

- '`collect2`'

  '`collect2`' is a utility to arrange calling of initialisation functions at start time. It links the program and looks for these functions, creating a table of them in a temporary '`.c`' file, and then links the program a second time but including the new file. '`ld`' is called from '`collect2`' at the end of this process.

NOTE

> Each of these processes is called with a number of flags that you'll not find in the '`gcc`' man pages. The reason for this is quite simple: they're flags specific to that process, and are nothing to do with `gcc`. In fact you can find out exactly how to use these flags by calling the process involved with `--help`, for example, when I invoke
>
> ```
> cc1 --help
> ```
>
> I get the full listing of the commands available for '`cc1`'. Although generally unnecessary to know these flags, it is helpful to know how to find out about the options available, which you may need to tweak and change.

Although `gcc` handles these details for you, it is worth getting a fuller grasp on what happens when `gcc` is invoked; the separate components enable you to monitor and watch each of the separate phases of compilation, such that (if necessary) you can cut out the text from a compile, change the flags and then paste the text back in and run it again with different flags.

The example compilation just shown was produced from a C source file; in practice, there is relatively little difference between this and, for example, the compilation of a Fortran source file compiled using `g77` (`g77` is the GNU Fortran compiler). The most obvious difference is what is used to compile the source file; '`cc1`' for C sources, '`f771`' for Fortran sources, '`jc1`' for Java sources etc..

4.4 Integrated Languages

So far the emphasis has been on the language C. Other languages, namely C++, Fortran, Objective-C and Java are also integrated into `gcc`. Our purpose here is to explain how to compile C++, Objective C, Fortran and Java source files (along with other notes regarding these languages). We'll also look at the various other front-ends supported by `gcc` in Section 4.4.6 [Other GCC Frontends], page 95.

For each of the languages discussed below, there is a frequently used options section and an example compilation section (showing the output of the compilation of a trivial source file in the old-time favourite example *Hello, world!* program).

4.4.1 How GCC deals with languages

There are a number of different ways to invoke `gcc` with regard to compiling different languages. The most obvious is to call the binary responsible for that language; `g++` for C++, `g77` for Fortran source files etc..

However, since all of these languages are integrated into `gcc`, it is possible to simply call `gcc` along with a number of flags telling it what language is being used. There are two things you'll need to be aware of: linking with the correct library for that language using the `-l` option, and supplying '`gcc`' with the correct language using the `-x` flag.

For example, invoking

```
$ gcc -lg++ -x c++ main.cpp
```

is exactly the same as calling

```
$ g++ main.cpp
```

for a C++ source file named '`main.cpp`'. However, although you can use the former, the latter is much easier to look at. The former just says "use the library '`libg++.so`', and tell '`gcc`' that the source file passed in is a C++ file".

However, we'll not consider supplying any information regarding language options or correct libraries to be passed to '`gcc`' (other than for Objective C); rather, we'll call the files directly, like `g++` for C++ source files, `gcj` for Java sources etc..

4.4.2 Objective C

There isn't too much to know about compiling Objective C sources; you must ensure that you link the program with the '`libobjc.a`' using `-lobjc`. Here's a few sources continuing the 'Hello, world!' theme - the first source file is '`main.m`':

```
/* main.m */
#include "HelloWorld.h"

int main()
{
  id helloWorld;
  helloWorld = [HelloWorld new];
  [helloWorld world];
  return 0;
}
```

And the header file containing the interface for the `HelloWorld` class:

```
/* HelloWorld.h */
#ifndef _HI_WORLD_INCLUDED
#define _HI_WORLD_INCLUDED
#include <objc/Object.h>
#define world_len 13

@interface HelloWorld : Object
{
  char helloWorld[world_len];
```

```
}
- world;

@end

#endif
```

And finally, the implementation of `HelloWorld`:

```
/* HelloWorld.m */
#include "HelloWorld.h"

@implementation HelloWorld

- world
{
  strcpy(helloWorld, "Hello, world!");
  printf("%s\n", helloWorld);
}

@end
```

Compiling this using the command

```
gcc -lobjc -lpthread -o helloWorld main.m HelloWorld.m
```

creates the executable '`helloWorld`' which, as you can predict, prints out the message "Hello, world!". You can use all the standard compilation options already described to compile Objective C programs.

NOTE You'll notice that I added `-lpthread` too. I did this, because if I didn't, the following errors popped up:

```
/usr/lib/gcc-lib/i486-suse-linux/2.95.2/libobjc.a
(thr-posix.o):
In function `__objc_init_thread_system':
thr-posix.o(.text+0x41): undefined reference to
`pthread_key_create'
/usr/lib/gcc-lib/i486-suse-linux/2.95.2/libobjc.a
(thr-posix.o):
...
In function `__objc_mutex_trylock':
thr-posix.o(.text+0x1f1): undefined reference to
`pthread_mutex_trylock'
collect2: ld returned 1 exit status
```

This is because without the `-lpthread` flag, a number of references from the archive '`libobjc.a`' could not be resolved, so we had to explicitly call in the `pthread` objects.

4.4.3 C++

The front-end for C++ is `g++`. `g++` is just a wrapper script to call `gcc` with C++ options. C++ is a super-set of C; thus, all C options will hold for a C++ compile, and they should (usually) be enough.

You have a number of choices when compiling C++ sources: invoke `g++`, which is the easiest option. The other option is to invoke `gcc` calling the correct language and libraries, for example `g++ -x c++ -lg++`, a little more verbose perhaps.

Not all of the options for compiling with `g++` are given here; many have been left out, and we'll only look at a small selection. The options that `gcc` offer should generally be enough for most compilations, except under very special curcumstances. You should consult the documentation for more details.

`-fdollars-in-identifiers`	Allow identifiers to contain `$` characters in identifiers. By default this shouldn't be permitted in GNU C++, although it is enabled on some platforms. `-fnodollars-in-identifiers` disables this option if it is default on the machine you're using.
`-fenum-int-equiv`	Allow conversion of `int` to `enum`.
`-fnonnull-objects`	This option ensures that no extra code is generated for checking whether any objects reached through references are not null.
`-Woverloaded-virtual`	Warn when a function in a derived class has the same name as a virtual function in the base class, but the signature is different.
`-Wtemplate-debugging`	When using templates, warn if debugging is not available.

4.4.4 Java

The front-end for Java is the GNU Java Compiler, `gcj`. Not all of the classes for Java have been implemented, which isn't surprising due to the impressive amount of classes that Java contains. In particular, support for the `Abstract Windowing Toolkit` and `Swing` components have not yet been implemented[8], and there is no support for `Remote Method Invocation` either. The Java FAQ `http://gcc.gnu.org/java/faq.html` details the reasons for current extent of support. It is worth visiting this site because new developments are made all the time as new classes and features are added. Also, the current status of the `gcj` compiler can be viewed at `http://gcc.gnu.org/java`.

The `gcj` compiler can take input files in the form of '`.java`' or '`.class`' files. If a '`.java`' source file is passed in, then it can either be passed to `gcj` and compiled into a '`.class`' file or compiled into native machine code. If `gcj` is passed a '`.class`' file, then it can only produce native machine code.

To view the machine-dependant classes in object-file format, supported by `gcj`, locate the '`libgcj.a`' archive library (mine is located in '`/usr/lib/`') and perform an **ar t libgcj.a** on it[9]. The results should be something like this:

```
...
EnumerationChain.o                BufferedInputStream.o
BufferedOutputStream.o            BufferedReader.o
BufferedWriter.o                  ByteArrayInputStream.o
ByteArrayOutputStream.o           CharArrayReader.o
CharArrayWriter.o                 CharConversionException.o
```

[8] That is up to the point when this book went to press.

[9] `ar` is an archiving tool for archives.

DataInput.o
DataInputStream.o
DataOutput.o
DataOutputStream.o
EOFException.o
File.o
FileDescriptor.o
FileInputStream.o
FileNotFoundException.o
FileOutputStream.o
FileReader.o
FileWriter.o
FilenameFilter.o
FilterInputStream.o
FilterOutputStream.o
FilterReader.o
FilterWriter.o
IOException.o
InputStream.o
InputStreamReader.o
InterruptedIOException.o
LineNumberInputStream.o
LineNumberReader.o
OutputStream.o
OutputStreamWriter.o
PipedInputStream.o
PipedOutputStream.o
PipedReader.o

...

which shows the output of a small section of the archive library. The Java-equivalent files are in object file format, obviously, since `gcj` takes java source code (or class files) and can convert the information into machine-dependent binary files.

GCJ Options

Let's look at some of the commands first, and then focus on a few relatively simple examples.

`gcj` is invoked as follows:

`gcj [options] file1 [[file2] ... [filen]]`

There are a number of different options available.

`-C` takes the '`.java`' file and produce a corresponding '`.class`' file. The '`.class`' file is in (machine independent) byte code, so it can be run with the JVM as normal. This option cannot be used with `--main=File` (see below), because no machine-dependent code is being produced. Thus, the command **gcj -C SomeClass.java** will produce a file named '`SomeClass.class`' which can be run with the `java` command as normal.

`--main=File` specifies the file to be used when searching for the file that would normally be invoked with `java [filename].java` for the file named '`[filename].class`' where the `main` method is specified. Cannot be used with `-C`. You *must* specify this when creating a binary produced in native format, because the usual `main` stub cannot be found unless we tell it where to look; remember that Java can specify a `public static void main(...)` method in any of its classes, so we need some way of telling the compiler where `main` will be located.

`-o file` creates the executable named '`file`' instead of the default '`a.out`'. This flag cannot be used in conjuction with `-C`, because you cannot rename the class file (Java '`.class`' files are named according to their corresponding '`.java`' source files).

`-d dir` places `class` files in directory '`dir`'. Only used when compiling bytecode using `-C`.

`-v` As with `gcc`, print verbose output to '`stdout`'.

`-g` Produce debugging information, when creating machine dependent code; this is useful since it enables you to use (for example) `gdb` or `ddd` to debug java source files that have been made into machine-dependent binaries - see Chapter 14 [Debugging with gdb and DDD], page 247.

Compiling a simple Java source file

The following example utilises two Java source files.

```
$ cat Main.java
/* Main.java */
import helloworld.HelloWorld;

public class Main
{
    public static void main(String args[])
    {
      HelloWorld helloWorld = new HelloWorld();
      System.out.println(helloWorld.toString());
    }
}
$
```

and our HelloWorld `package` file located in the directory '`helloworld`' in the same directory that '`Main.java`' is in:

```
$ cat helloworld/HelloWorld.java
/* HelloWorld.java */
package helloworld;

public class HelloWorld
{
  String helloWorld;
  public HelloWorld()
  {
    helloWorld = "Hello, world!";
  }
  public String toString()
  {
    return helloWorld;
  }
}
$
```

Compiling this program with the command

```
$ gcj --main=Main Main.java helloworld/HelloWorld.java -o HelloWorld
```

yields the following output (if any problems occur, it may be because your classpath may not be set. If you can't fix it by using any of the options already given, try looking at the FAQ at the `gcj` homepage at `http://gcc.gnu.org/java/faq.html` for answers to many common problems when trying to compile java programs):

```
$ ./HellowWorld
Hello, world!
$
```

'Helloworld' is our machine-dependent binary. If we wanted, we could have simply produced a '.class' file

```
$ gcj -C Main.java helloworld/HelloWorld.java
```

Which produces 'Main.class' and 'HelloWorld/HelloWorld.class', which can be ran using the command java Main:

```
$ java Main
Hello, world!
$
```

4.4.5 Fortran

The front end compiler for Fortran is g77. It is used to compile GNU Fortran programs; however, other Fortran dialects are also supported by a number of flags. For details of the GNU Fortran language, refer to http://gcc.gnu.org/onlinedocs/g77_toc.html.

Compiling a simple fortran source file

As with many of the other front-ends, you can use many of the C language options such as -o, -g, -v etc. The options here should be enough for you to compile many Frotran programs, although you should refer to the 'g77' documentation for a broader range of flags.

g77 Options

-ffree-form, -fno-fixed-form	by default, compilation will vi for fixed form Fortran code, based on punched-card format. Specifying -ffree-form or -fno-fixed-form allows compilation of the new Fortran 90 free form source code.
-ff90	allow *some* Fortran 90 constructs; not all may be supported, depending on current support for the compiler you're using.
-I-, -IDIR	Files included by the Fortran INCLUDE directive are not preprocessed; thus, use -IDIR to search for INCLUDE files in directory 'DIR'. Do *not* put a space between the switch and the directory.
-x f77-cpp-input	Ensure that the source file is preprocessed by the preprocessor, 'cpp'. This enables you to pass -D options to the preprocessor inside the Fortran file (see Section 4.2.8.2 [Defining Constants], page 83), as well as be able to deal with #ifdef and #if statements etc. in your code.

Like the previous section on gcj, we'll illustrate a simple hello world program, outlined below in the new Fortran *free form* format, with a few preprocessor options in there:

```
$ cat HelloWorld.for
      PROGRAM HELLOWORLD
#if HELLO
      WRITE(6,*) 'Hello, world!'
#else
      WRITE(6,*) 'Goodbye, world!'
#endif
      END PROGRAM HELLOWORLD
$
```

I compiled this program using the command g77 -x f77-cpp-input -DHELLO -ffree-form HelloWorld.for

You can well imagine the output of this program - so we'll not bother listing it. If we'd not have included the `-DHELLO` definition, the output would have been instead `"Goodbye, world!"`.

4.4.6 Other GCC Frontends

A number of languages are supported; below is a list of current front-ends:

- The GNU Ada Translator (GNAT)
- GNU Pascal Compiler (GPC).
- Mercury
- Cobol For GCC
- G95 (Fortran 95)
- GNU Modula-2
- Modula-3

However, for up-to-date information, visit `http://gcc.gnu.org/frontends.html`.

4.5 Pulling it Together

You have seen the basic compilation options on offer, so let's concentrate on some practical examples. This section will utilise the M4 sources, to give you something tangible to play with. Not all of the options will be used from the preceeding sections - however, I'll utilise as many as necessary to give you a good grounding in `gcc`'s options.

The following examples are split up into the main phases of compilation, although many of the other commands will also be used within these sections.

4.5.1 Preparation

To begin with, you'll need the M4 sources. Once you have installed them you'll end up with a directory where the binaries are kept, and the original source directory: don't delete this, as we'll use the source files to demonstrate different `gcc` options. By default when you run `./configure`, `make` etc., the libraries and object files are put into the source tree. So after I'd installed M4 from my '`$HOME/gnu-src/m4-1.5`' directory, the libraries were placed in the source tree under '`$HOME/gnu-src/m4-1.5/m4/.libs`':

```
$ ls $HOME/gnu-src/m4-1.5/m4/.libs
libm4.a  libm4.la  libm4.lai  libm4.so  libm4.so.0  libm4.so.0.0.0
$
```

These libraries are built from the '`m4`' directory of sources. The files that we'll actually be interested in utilizing are in the '`src`' directory of M4:

```
$ ls $HOME/gnu-src/m4-1.5/src
freeze.c getopt.c m4.h main.c stackovf.c
$
```

NOTE In addition you'll need a temporary file in this directory (I named mine '`temp.c`') with the following details:

```
#include "m4module.h"
const lt_dlsymlist *lt_preloaded_symbols = 0;
```

This is because normally `libtool` would handle this for you; since we are hand-running the examples (yes, it may seem counter-intuitive, but it's

the only reasonable way to illustrate the example) we have to generate the file instead.

4.5.2 Preprocessing

Let's start with preprocessing. Locate the file '`main.c`' in the '`m4/src`' directory. Send it to the preprocessor using the command

```
gcc -E main.c -I../m4/ -I. -I..
```

which will display the result straight to screen. I've included the `-I` options because '`main.c`' includes a number of files in different directories. Since this command will send the output straight to the screen, to be able to have a look at the output, I sent it to a temporary file ('`temp.i`') using file redirection:

```
gcc -E main.c -I../m4/ -I. -I.. > temp.i
```

Check out the file sizes now you've preprocessed the file. There is a big difference; mine showed the following sizes (the sizes will differ, possibly radically, depending on which build version you use):

```
$ ls -l main.c temp.i
-rw-r--r--   1 rich     users       13995 Jun  2 01:12 main.c
-rw-r--r--   1 rich     users       58376 Aug  1 16:53 temp.i
```

Notice that it's almost five times bigger. This is because the `#include`s have packed in all their information. We can study this by looking at a `grep` of the preprocessed file, filtering for all lines beginning with a `#` symbol:

```
$ grep '^#' temp.i
# 1 "main.c"
# 1 "/usr/include/getopt.h" 1 3
# 160 "/usr/include/getopt.h" 3
# 20 "main.c" 2
# 1 "/usr/include/signal.h" 1 3
# 1 "/usr/include/features.h" 1 3
# 138 "/usr/include/features.h" 3
# 196 "/usr/include/features.h" 3
# 1 "/usr/include/sys/cdefs.h" 1 3
...
# 1 "../m4/m4module.h" 1
# 1 "../m4/error.h" 1
# 1 "../m4/system.h" 1
# 100 "../m4/system.h"
# 143 "../m4/system.h"
# 27 "../m4/error.h" 2
# 23 "../m4/m4module.h" 2
# 1 "../m4/list.h" 1
# 24 "../m4/m4module.h" 2
# 1 "../m4/ltdl.h" 1
# 75 "../m4/ltdl.h"
# 107 "../m4/ltdl.h"
# 278 "../m4/ltdl.h"
# 301 "../m4/ltdl.h"
# 25 "../m4/m4module.h" 2
# 1 "/usr/include/obstack.h" 1 3
# 338 "/usr/include/obstack.h" 3
```

```
...
```

If you actually look at the contents of '`temp.i`', there will be massive patches of white space; this occurs when the preprocessor is stripping out comments that do not need to be included (and if included would make the file even larger). Recall from Section 4.2.3 [The Preprocessor], page 79 that the above `#`'s have a special significance; they tell us where and when a file is included, and which files are included from the file being included (and so on). For example, one portion of my '`temp.i`' file read as follows (omitting newlines where necessary to save space):

```
# 227 "/usr/include/signal.h" 2 3
...
extern int sigprocmask  (int __how,
     __const sigset_t *__set, sigset_t *__oset)    ;

extern int sigsuspend  (__const sigset_t *__set)    ;

extern int __sigaction  (int __sig, __const struct sigaction *__act,
     struct sigaction *__oact)    ;
extern int sigaction  (int __sig, __const struct sigaction *__act,
   struct sigaction *__oact)    ;
...
# 1 "/usr/include/bits/sigcontext.h" 1 3
```

The fist line, `# 227 "/usr/include/signal.h" 2 3` is saying "At line number 227 in '`/usr/include/signal.h`', note the return to this file from some other file, and supress any warnings that may arrise from the following textual substitution". Then, at the end of the above code snippet, '`/usr/include/bits/sigcontext.h`' is included, and off the preprocessor goes again.

4.5.3 Compilation

After preprocessing, our next goal is produce assembly language for the preprocesed file. There are a number of approaches to this; we could invoke the preprocessor on each '`.c`' file, produce a '`.i`' file and then pass this to the compiler; or (a lot less trouble) just pass the '`.c`' files to the compiler, which will detect that the sources have not been preprocessed, so instead will preprocess them and then run them through the compiler:

```
gcc -S -I ../m4 -I ../ main.c freeze.c stackovf.c temp.c
```

Note that this will produce a number of errors - mine produced the following output:

```
$ gcc -I. -I.. -I../m4 -S main.c freeze.c stackovf.c temp.c
In file included from main.c:23:
m4.h:60: parse error before ‘malloc’
m4.h:60: warning: data definition has no type or storage class
m4.h:61: parse error before ‘realloc’
m4.h:61: warning: data definition has no type or storage class
main.c: In function ‘main’:
main.c:239: ‘STDIN_FILENO’ undeclared (first use in this function)
main.c:239: (Each undeclared identifier is reported only once
main.c:239: for each function it appears in.)
main.c:367: ‘PACKAGE’ undeclared (first use in this function)
main.c:367: ‘VERSION’ undeclared (first use in this function)
In file included from freeze.c:23:
m4.h:60: parse error before ‘malloc’
m4.h:60: warning: data definition has no type or storage class
```

```
m4.h:61: parse error before ‘realloc’
m4.h:61: warning: data definition has no type or storage class
freeze.c: In function ‘produce_frozen_state’:
freeze.c:205: ‘PACKAGE’ undeclared (first use in this function)
freeze.c:205: (Each undeclared identifier is reported only once
freeze.c:205: for each function it appears in.)
freeze.c:205: ‘VERSION’ undeclared (first use in this function)
In file included from stackovf.c:80:
m4.h:60: parse error before ‘malloc’
m4.h:60: warning: data definition has no type or storage class
m4.h:61: parse error before ‘realloc’
m4.h:61: warning: data definition has no type or storage class
$
```

The reason is that a number of symbols - such as `STDIN_FILENO` and `VERSION` have not been resolved; they were not found in any of the header files that we specified using `-I`. This is because we'll actually reslove these symbols in the last stage (linking) by including the relevant libraries. Despite these errors, we now have a collection of '`.s`' files:

```
$ ls -l *.s
-rw-r--r--   1 rich     users       44080 Sep 30 18:25 freeze.s
-rw-r--r--   1 rich     users       20096 Sep 30 18:25 main.s
-rw-r--r--   1 rich     users         810 Sep 30 18:25 stackovf.s
-rw-r--r--   1 rich     users         949 Sep 30 18:25 temp.s
$
```

4.5.4 Assembling

Now we have the assembled sources, we can perform the penultimate stage and produce object files for each '`.s`' file.

```
gcc -DHAVE_CONFIG -I. -I.. -I../m4 -c main.s freeze.s stackovf.s temp.s
```

which results in the corresponding object files being produced:

```
-rw-r--r--   1 rich     users       13108 Sep 30 18:59 freeze.o
-rw-r--r--   1 rich     users       12796 Sep 30 18:59 main.o
-rw-r--r--   1 rich     users        1528 Sep 30 18:59 stackovf.o
-rw-r--r--   1 rich     users        1565 Sep 30 18:59 temp.o
```

4.5.5 Linking

Finally, we're ready to link the files together to produce a final binary. This involves taking the object files and glueing them together with the M4 library, '`libm4.so`' to produce a binary that we can run. To make sure that your M4 libraries are visible, point them to your '`lib`' directory:

```
export LD_LIBRARY_PATH
LD_LIBRARY_PATH=$HOME/gnu-src/m4-1.5/m4/.libs:$LD_LIBRARY_PATH
```

Finally, run the object files through `gcc`:

```
gcc -DHAVE_CONFIG_H -I. -I.. -I../m4 main.o freeze.o stackovf.o temp.o -L‘cd
../m4/.libs && pwd‘ -lm4
```

which produces the executable '`a.out`'. Let's just test it:

```
$ ./a.out –version
GNU (null) (null)
```

```
Written by Rene' Seindal and Gary V. Vaughan.

Copyright 1989-1994, 1999, 2000 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is
NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
$
```

4.5.6 And Finally...

That all seemed like a lot more work than we really needed. In fact, it *was* a lot more work; we could have simply called

`gcc -DHAVE_CONFIG_H -I. -I.. -I../m4 main.c freeze.c stackovf.c temp.c -L`cd ../m4/.libs && pwd` -lm4`

which would have given us the same results, quicker than having to produce assembler sources, and then passing these on to the assembler, and so on.

In fact, we cheated from the start. M4 had to be built (so that the libraries were all in place, produced from the '`m4/m4`' and '`m4/ltdl`' directories). All we did is go in and compile a few sources and link them with these libraries. A few things to note:

- `gcc` was used directly to build the sources, and
- the libraries were already built *for* us.

Is there an easier way?

Actually, there is, and it ties in to Chapter 5 [GNU Make], page 103, Chapter 9 [Autoconf], page 189, Chapter 10 [Automake], page 203 and Chapter 11 [Libtool], page 205. GNU Make enables us to place all the information we need to build sources into neatly modularised '`make`' files. It saves us the hassle of typing in long-winded commands. Autoconf and automake enable us to configure and manage make files in a very simple fashion. Libtool makes library creation and management much simpler as well.

4.6 Reference Section

This section gives an overview of all of the commands used throughout this chapter regarding `gcc`. These commands are an extremely small subset of the full set of `gcc` commands. You should consult the `gcc` documentation for the comprehensive listing.

4.6.1 Standard Compilation Options

`gcc [options] file [file2] ... [filen]` Compile file '`file`' (and/ or ['`file2`'] ... ['`filen`']); preprocess, compile, assemble and link, where `[options]` can be any of the options below:

`-o outputFile`	Replace default file with file named '`outputFile`'. This can be used with any stage that produces as output some default file. For example, `gcc -S main.c -o file.assembler` will produce an assembly file named '`file.assembler`' instead of the default '`main.s`'.
`-E`	Preprocess only; send to '`stdout`' (standard output).
`-C`	Preserve line comments; used with `-E`.

`-P`	Do not generate `#line` comments; used with `-E`.
`-S`	Preprocess (if the file is a C source file) and compile (if the file is a preprocessed file), but do not assemble. The created file will have the suffix '`.s`', unless the `-o 'outputFile'` option is used.
`-c`	Preprocess (if the file is a C source file), compile (if the file is a preprocessed file) and assemble (if the file is assembler source); do not link. The created file will have the suffix '`.o`', unless the `-o 'outputFile'` option is used.
`-traditional`	Supports the traditional C language. The traditional option also supports all of the FSF's extensions to the C language.
`-ansi`	Supports the ANSI C standard.
`-pedantic`	Issues all the warning messages that are required by the ANSI C standard. Forbids the use of all the FSF extensions to the C language and considers the use of such extensions errors.
`-DDEFN`, `-DDEFN=VALUE`	Define `DEFN` to have value 1, or `DEFN` to have value `VALUE`.
`-UDEFN`	undefine definition `DEFN` (all `-D` options are evaluated before any `-U` options on the command line).
`-v`	Print verbose information.
`-pipe`	Use a pipe rather than temporary files when compiling.
`-Idirectory`	When searching for '`.h`' header files, look in directory '`directory`', in addition to the default directories. If spedified before the `-I-` option (see below), only `#include "..."` files are searched for. Used after `-I-`, `-I 'directory'` will also search for any '`#include <...>`' files.
`-I-`	Used after the `-I` option, it forces all *succeeding* `-I` calls to search for `#include <...>` files. It also negates search of the current directory; thus, to force the compiler to look in the current directory, use `-I.` after `-I-`.
`-Wa,option-list`	Pass *option-list* as a list of comma separated options (with no white-space between options whatsoever) to the assembler.
`-Wl,option-list`	Pass *option-list* as a list of comma separated options (with no white-space between options whatsoever) to the linker.

4.6.2 Linking and Libraries

`-lname`	Include library '`libname.so`' if it exists; if '`libname.so`' doesn't exist, search for '`libname.a`'.
`-Ldirectory`	search directory '`directory`' for library files. Used with `-l`.
`-static`	indicate that static libraries should be linked *instead* of shared libraries.

4.6.3 Warning Options

Since the warning options are fairly descriptive on their own (apart from the `-w` option, which inhibits all warning options), they're listed here without any descriptions; you can find out what each of them do in Section 4.2.9 [Warnings], page 85.

`-pedantic`, `-pedantic-errors`, `-Wall`, `-Wimplicit-int`, `-Wimplicit-function-declaration`, `-Wimplicit`, `-Wmain`, `-Wreturn-type`, `-Wunused`, `-Wswitch`, `-Wcomment`, `-Wformat`, `-Wchar-subscripts`, `-Wuninitialized`, `-Wparentheses`

4.6.4 Language Options

4.6.4.1 Objective C Command Summary

Other than linking your Objective-C sources with '`libobjc.a`' using the `-l` option, there aren't any other options other than the standard options already discussed for C that you need to be aware of.

4.6.4.2 C++ Command Summary

Generally you can compile many C++ source files using the standard compilation options already discussed for C. There are a few extra options here that you may find useful:

`-fdollars-in-identifiers` — Allow identifiers to contain `$` characters in identifiers. By default this shouldn't be permitted in GNU C++, although it is enabled on some platforms. `-fnodollars-in-identifiers` disables this option if it is default on the machine you're using.

`-fenum-int-equiv` — Allow conversion of `int` to `enum`.

`-fnonnull-objects` — This option ensures that no extra code is generated for checking whether any objects reached through references are not null.

`-Woverloaded-virtual` — Warn when a function in a derived class has the same name as a virtual function in the base class, but the signature is different.

`-Wtemplate-debugging` — When using templates, warn if debugging is not available.

4.6.4.3 Java Command Summary

`gcj [options] file1 [[file2] ... [filen]]`

Where `[options]` options can include the following:

`-C` — take the '`.java`' file and produce a corresponding '`.class`' file. Cannot be used with `--main=HelloWorld`.

`--main=File` — Specify the name of the '`.java`' file to locate the `public static void main(String args[])` method as '`File`'. Cannot used with `-C`.

`-o file`	create the executable named '`file`' instead of the default '`a.out`'. This flag cannot be used in conjuction with `-C`, because you cannot rename the class file.
`-d dir`	place `class` files in directory '`dir`'. Only used when compiling bytecode using `-C`.
`-v`	Print verbose output to '`stdout`'.
`-g`	Produce debugging information.

4.6.4.4 Fortran Command Summary

`g77 [options] sourcefile`

compile '`sourcefile`', providing it is fixed-form format, where `[options]` can be any of the options below:

`-ffree-form`	use free-form format.
`-fno-fixed-form` `-ff90`	Support *some* of the constructs available in Fortran 90.
`-I-`, `-IDIR`	Files included by the Fortran `INCLUDE` directive are not preprocessed; thus, use `-IDIR` to search for `INCLUDE` files in directory '`DIR`'.
`-x f77-cpp-input`	Ensure that the source file is preprocessed by the preprocessor, '`cpp`'.

4.7 Summary

The purpose of this chapter was to introduce `gcc` at a non-complex, yet useful, level. The examples were not too demanding, and covered a broad spectrum of commonly used options.

The examples throughout this chapter revolved around hand-typing commands in a terminal. For very small projects of only a few files, this is fine; larger projects require much more attention. The next chapter, GNU Make, takes a look into using '`make`' files for projects, to enable you to easily manage how projects are built.

`gcc` is one of the fundamental tools for building software; but for advanced projects, advanced tools are needed. Although GNU make is useful for source code management, Chapter 9 [Autoconf], page 189 and Chapter 10 [Automake], page 203 enable us to extend make files auch that configuration and management of make files becomes much easier. In addition, Chapter 11 [Libtool], page 205 makes library creation and management much simpler too.

5 Automatic Compilation with Make

In the last chapter, we explained how to invoke GCC to compile your source code into an executable. As a project grows, and the source files multiply, the time taken to compile and link them all whenever some of the source files have been edited becomes a nuisance. The GNU environment provides an implementation of the UNIX Make utility to help you to manage the process of rebuilding programs from the source files they depend upon automatically.

The next section gives an overview of what Make is, and the problems that it wants to solve for you. The rest of the chapter is a discussion of how to utilise Make, and the sorts of things you might want to have in your own '`Makefile`'s. If you already know about Make, you can skip this chapter and refer back to it when you read about GNU Automake later (see Chapter 10 [Automake], page 203).

GNU Make is a sophisticated tool with many features beyond those described in this chapter, and enhancements beyond the standard feature set offered by the original UNIX Make upon which it is based. In this book we will describe only the basic features of GNU Make: sufficient to understand how it works, and to extend the basic configuration of GNU Automake (see Chapter 10 [Automake], page 203). If you want to learn more about the Make utility, there are some suggestions in Section 5.8 [Further Reading], page 125. However, when you develop a project with Automake and the other GNU utilities described in this book, there is no need to use (or even learn about) anything beyond what we describe here.

5.1 The Make Utility

Fundamentally, the Make utility looks for files, and uses them to make other files. Although it has many uses beyond those we describe here, in this book we are interested in Make as a tool for development – to compile, install and test programs and libraries.

That is, Make is a supremely flexible tool that is useful to us at most stages of the development process in one way or another. It is also routinely deployed for non development oriented tasks: the UNIX NIS[1] facility uses Make to keep its databases up to date, and push modified tables over the network.

Make follows the long established UNIX mantra: "Do *one* job, and do it well". The *one job* that Make does is to keep files up to date.

5.1.1 Targets and Dependencies

As with all good explanations of UNIX facilities, we will start with **Hello World**, in C:

[1] Network Information Service. Disseminates a centralised database of network details, such as hostnames and user password details, through a network.

```
#include <stdio.h>

int main (int argc, const char *argv[])
{
    printf ("Hello, World!\n");
    return 0;
}
```

Example 5.1: *'hello.c' – an old favourite*

In a directory that contains nothing but this one file, you can invoke `make` to make a program out of it, like so:

```
$ make hello
gcc    hello.c   -o hello
$ ./hello
Hello, World!
```

The program '`hello`' that Make has been asked to build is known as the *target*. The files that are required in order to bring the target up to date, or to *refresh the target*, are called the target's *dependencies*[2].

Make knows an awful lot about the compilation process, and the relationships between file names of related *suffixes* (also known as *extensions*). In the example above, when we asked `make` to refresh '`hello`' for us, it was able to infer that it should compile '`hello.c`', in part due to the presence of '`hello.c`', but also due to the fact that files with a '`.c`' suffix are normally passed to the C compiler. This inference depends on the association between the target name, '`hello`', and the source file name, '`hello.c`'; Make did not merely pick '`hello.c`' because it was the only file in the build directory. For example in the same directory, it won't work if we try to `make` a target with no matching source file name:

```
$ make helloagain
make: *** No rule to make target 'helloagain'.  Stop.
```

Internally, Make contains an extensive database of commands that are normally used to transform between files with names that differ only in suffix. As another example, Make knows that intermediate compilation objects are held in files with a '`.o`' suffix (see Chapter 4 [The GNU Compiler Collection], page 75), and that the C compiler performs that transformation:

```
$ make hello.o
gcc    -c -o hello.o hello.c
```

5.1.2 A Refreshing Change

Near the start of this chapter we stated that *Make keeps files up to date*. In each of the examples so far, we have given Make a target that it must bring up to date: '`hello`', '`helloagain`' and '`hello.o`'. On each occasion, Make has found that there is no such file and tried to determine a way to create one from its inference model. If a target is *already* up to date with respect to its dependencies then Make will notice, and not go to the trouble of refreshing it:

[2] Other books refer to the dependencies as *prerequisites*; the two terms are interchangable.

```
$ ls -ltr
total 9
-rw-r--r--   1 gary     users       100 Sep 16 13:40 hello.c
-rwxr-xr-x   1 gary     users      5964 Sep 16 13:41 hello
-rw-r--r--   1 gary     users       888 Sep 16 13:43 hello.o
$ make hello.o
make: 'hello.o' is up to date.
$ make hello
gcc   hello.o   -o hello
$ ls -ltr
total 18
-rw-r--r--   1 gary     users       100 Sep 16 13:40 hello.c
-rw-r--r--   1 gary     users       888 Sep 16 13:43 hello.o
-rwxr-xr-x   1 gary     users      5964 Sep 16 13:44 hello
$ ./hello
Hello, World!
```

Make is using the modification timestamp of these file to decide whether the target file is out of date with respect to the files it depends on. Here, '`hello.c`' has not been changed since '`hello.o`' was last refreshed, so when asked to '`make hello.o`', Make doesn't recompile. The program '`hello`' is actually older than '`hello.o`' to start with, so when we ask for '`hello`' to be refreshed, Make does indeed relink it from the newer '`hello.o`'.

The strategy used by Make for refreshing targets depends on the files that are present in the build directory. Originally, with only '`hello.c`' available, Make compiled directly from source to the program '`hello`', but in the previous example where a '`hello.o`' object file was also available, Make took the simpler route of relinking '`hello`' directly from the object file. There is no magic involved here: the search order through Make's database of candidate file name suffixes tries less processor intensive matches first. This is important when it comes to complex builds, where Make is able to minimise the amount of recompilation that is performed by making the best use of intermediate files.

Make was developed in the early days of UNIX to automate the task of recompiling applications from of source files and libraries. Before Make was invented developers had to use shell scripts to recompile everything whenever an application was rebuilt.

5.2 The Makefile

If you imagine applying what we have shown you of Make so far to a real project, some limitations will quickly come to light, not least of which is that your program is almost certainly not compiled from a single similarly named source. The behaviours we have demonstrated so far are merely the last line defaults. In practice, the build process is codified in a '`Makefile`'. In this '`Makefile`', you list the relationships between the sources of your project along with some transformation rules to specialise Make's default database to meet the project's build requirements.

Here, *example 5.2* shows a small Makefile for building the program '`m4`' using three object files, '`main.o`', '`freeze.o`' and '`stackovf.o`', and a library from another directory. The '`libm4.a`' library declared in '`m4_LDADD`' is built from its own source code by another Makefile in the library's own build directory.

```
m4_OBJECTS = main.o freeze.o stackovf.o
m4_LDADD   = ../m4/libm4.a

m4: $(m4_OBJECTS)
        $(CC) -o $ $(m4_OBJECTS) $(m4_LDADD)
clean:
        rm -f $(m4_OBJECTS)
# Object compilation rules follow
main.o: main.c
        $(CC) -c main.c

freeze.o: freeze.c
        $(CC) -c freeze.c


stackovf.o: stackovf.c
        $(CC) -c stackovf.c
```

Example 5.2: *A simplistic* `'Makefile'`

This file is actually longer than it needs to be, since some of the text included here explicitly (for the purpose of illustration) is supplied from Make's own default internal rules. Although the file might seem somewhat indigestable at first, in broad terms a Makefile can contain only three different structures: Rules, variables and comments. We will describe each in turn over the following subsections, and then move on to some specialisations of the basic Makefile building blocks for the rest of the chapter.

5.2.1 Make Rules

In order for Make to help with your project, you must tell it about the relationships between all of the files it will be maintaining for you. Each relationship is expressed as a *dependency rule* and takes the following form:

```
target: dependency ...
        command
        ...
```

Example 5.3: *Format of a dependency rule*

Here, *target* relates to a file that needs to be built during the compilation phase of your project, for example '`hello.o`'. A space delimited list of *dependency* files name every other file that needs to be up to date before *target* can be built. And an indented list of *command* each comprise a line of shell script code for Make to run when *target* needs refreshing. You might say that Make uses *command* to build *target* from *dependency*.

NOTE Make will understand that each *command* is part of the preceding rule only if the first character on each line is a literal tab. Although a string of spaces, or even a space followed by a tab look the same in most editors, they will cause an error like this when you try to run `make`:

```
$ make
Makefile:159: *** missing separator.  Stop.
```

Notice that GNU make tells you the name of and the line number within it that caused the error (line 159 of '`Makefile`' in this case), making it easy to locate and correct once you know what the cryptic error message means.

Recall that in the last chapter you saw that the following command will build the '`m4`' binary:

```
$ gcc -o m4 main.c freeze.c stackovf.c ../libs/libm4.a
```

The simplest rule to achieve the same effect using a '`Makefile`' is as follows:

```
m4: main.c freeze.c stackovf.c ../libs/libm4.a
        gcc -o m4 main.c freeze.c stackovf.c ../libs/libm4.a
```

Example 5.4: *A sample dependency rule*

With just this one rule in the '`Makefile`', recompiling '`m4`' is a simple matter of entering the command '`make m4`' at your shell prompt:

```
$ make m4
gcc -o m4 main.c freeze.c stackovf.c ../libs/libm4.a
$ make m4
make: 'm4' is up to date.
```

On the second invocation, `make` doesn't build '`m4`' again, because it is already newer than all of the dependencies listed in the rule.

Anyway, as we explained in Section 5.1.2 [A Refreshing Change], page 104, this isn't a good way to structure a project '`Makefile`', since absolutely everything is recompiled each time any file is changed. You can avoid such needless recompilations, and drastically cut down on the amount of work GCC has to do when recompiling after changing some source files, by breaking the compilation down into phases. Recall that source code can be compiled into object code files, which are in turn linked to form the target executable (see Chapter 4 [The GNU Compiler Collection], page 75). Each stage is represented by its own rule in your '`Makefile`', like this:

```
m4: main.o freeze.o stackovf.o ../lib/libm4.a
        gcc -o m4 main.o freeze.o stackovf.o ../lib/libm4.a

main.o: main.c
        gcc -c main.c

freeze.o: freeze.c
        gcc -c freeze.c


stackovf.o: stackovf.c
        gcc -c stackovf.c
```

Example 5.5: *Compilation in phases with make*

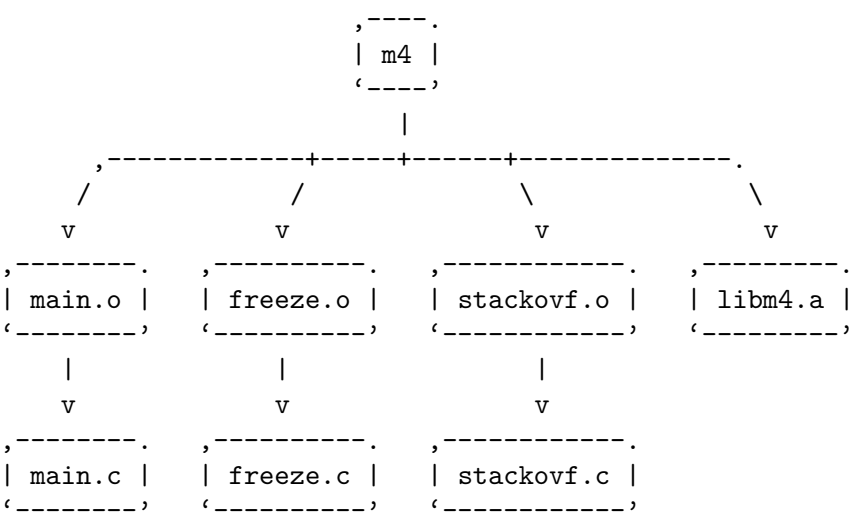
Now, with this '`Makefile`' in place, imagine that '`freeze.c`' has been edited since the compilation we just did with *example 5.4*. Invoking `make m4` now executes only the commands from the set of rules needed to refresh the '`m4`' target file:

```
$ make m4
gcc -c freeze.c
```

```
gcc -o m4 main.o freeze.o stackovf.o ../lib/libm4.a
```

Previously, `gcc` had needed to recompile all three source files, even if only '`freeze.c`' had actally changed. This can save literally hours when recompiling projects spread across a great many source files.

A typical '`Makefile`' will specify many rules. Together they can be represented as a tree of dependencies, where many of the dependencies for a particular target are themselves targets in another rule, often with more dependencies of their own. The root of this tree is the *target* that Make ultimately aims to bring up to date, and is normally specified on the command line. We used '`make m4`' above, which caused Make to start building the tree with the '`m4`' rule at its root, aiming to bring '`m4`' up to date.



Example 5.6: *Part of a Makefile dependency tree*

If `make` is invoked without any arguments, then the *target* of the first dependency rule in the '`Makefile`' is refreshed by default. To take advantage of this behaviour, it is common practice to name the first rule '`all`', and set it up to be dependent on all of the programs and libraries compiled by this '`Makefile`'. By doing this, invoking *`make all`* or even just *`make`* will refresh all of the targets managed by this '`Makefile`'. For example, in the directory where all of a project's plugin modules are compiled, the Makefile would contain an '`all`' target that depends on all of the plugin targets that are built by that Makefile:

```
all: gnu.so load.so m4.so traditional.so perl.so stdlib.so

gnu.so: $(gnu_so_OBJECTS)
        $(LD) $(LDFLAGS) -o gnu.so $(gnu_so_OBJECTS) $(gnu_so_LDADD)

load.so: $(load_so_OBJECTS)
        ...
```

Example 5.7: *A typical all target Makefile fragment*

Make can determine whether any *target* in its rules has gone out of date from the modification times of the files referred to in the *target* and *dependency* parts of all of its rules. Make first checks that each of the *dependency* files listed against the *target* do not have later modification times than the *target* itself. If any of them do, then Make executes the list of *command* lines

associated with the out of date *target*. However, the timestamp checks are performed recursively, so before it can determine whether the current *target* is older than any of its listed *dependency* files, it must first recursively ensure that each of those files is up to date with respect to their own *dependency* lists.

If you look at the *command*s in each rule from *example 5.5*, you will see that each *command* is written so that it builds a new *target* file from those files listed as its *dependencies*. Whenever *target* names a file, this statement is always true: *command* transforms *dependency* into *target*. If you bear this in mind as you create new rules, then choosing appropriate file names and commands becomes quite straight forward.

5.2.2 Make Variables

The next basic building block used when creating a '`Makefile`' is a variable assignment. Assigning a value to a variable[3] within a '`Makefile`' looks just as you would expect:

```
variable-name = value
```

Example 5.8: *Format of Make variable assignment*

Where *variable-name* is any string of characters not containing whitespace or other characters that are special to Make: '`=`' is used to separate a variable's name from its value, '`:`' is used to mark the end of a target name in a dependency rule, and '`#`' is used to begin a comment (see Section 5.2.3 [Make Comments], page 112), so you cannot use any of those three characters in the *variable-name* itself. Obvious problems with readability aside, if you want your '`Makefile`' to work with other implementations of Make, you should probably limit yourself to *variable-name*s made from upper and lower case letters, the digits '`0`' to '`9`' and the underscore character.

The *value* consists of the remaining characters up to the end of the line, or the first '`#`' character encountered – whichever comes first. Also, any leading or trailing whitespace around *value* does not become part of the variable's contents.

A variable assignment can extend across several lines if necessary by placing a '`\`' at the end of each unfinished line:

```
libm4_a_SOURCES         = builtin.c debug.c error.c eval.c hash.c \
                          input.c ltdl.c macro.c module.c output.c \
                          path.c regex.c symtab.c utility.c
```

Example 5.9: *Extending a Make variable assignment over several lines*

Whenever a continuation backslash is encountered in a variable declaration like this, the backslash itself and all of the whitespace that surrounds it – including all of the leading whitespace on the following line – is deleted and replaced by a single space as it is assigned to *variable-name*.

Make will expand variable references by using the *value* currently stored in the variable where ever the following syntax is used:

```
$(variable-name)
```

Example 5.10: *Format of Make variable expansion*

[3] Other text books refer to Make variables as *macros*, but that term has fallen out of favour. To a computer scientist, the word *macro* describes something quite different to the functionality of Make *variables*.

By convention, the name of any external command used in a '`Makefile`', and any option required by such a command is stored in a variable. For example, you would use `$(CC)` and `$(CFLAGS)` rather than writing '`gcc`' and '`-ggdb3`' directly into the rules. Rewriting our '`Makefile`' to employ variables looks like this:

```
CC       = gcc
CFLAGS   = -ggdb3
CPPFLAGS = -DDEBUG=1

m4_OBJECTS = main.o freeze.o stackovf.o
m4_LDADD   = ../lib/libm4.a

m4: $(m4_OBJECTS) $(m4_LDADD)
        $(CC) $(CFLAGS) -o m4 $(m4_OBJECTS) $(m4_LDADD) $(LIBS)

main.o: main.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c main.c

freeze.o: freeze.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c freeze.c


stackovf.o: stackovf.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c stackovf.c
```

Example 5.11: *Use of variables in a Makefile*

This provides an obvious advantage: you can change the contents of a variable in one place, but affect all of the rules that use the variable. *example 5.11* is set up to compile with debugging symbols ('`-ggdb3`'), and with additional debugging code enabled by the preprocessor ('`-DDEBUG=1`'). This is ideal while the code is under development. Later, when a project is ready to be released, it is very easy to change the variable assignments to compile a production version of the code. You simply change the values of these variables at their declaration near the top of the '`Makefile`':

```
CC       = gcc
CFLAGS   = -O2
CPPFLAGS =
...
```

Example 5.12: *Changing variable values in a Makefile*

Actually, Make's internal database contains default definitions for many of these variables already, and uses them for the *command* part of its default rules. The default setting for '`$(CC)`' was used earlier, when we compiled '`hello.c`' without a Makefile in Section 5.1.1 [Targets and Dependencies], page 103. If you make use of any variable in your own commands, you should not fall into the habit of relying on there being a useful default definition built in to Make – one of your customers will probably try to run your Makefile through a version of `make` that has a different value, which breaks the assumptions you made. If you use '`$(CPPFLAGS)`' in your own rules, define '`CPPFLAGS`' somewhere in your Makefile, even if there is nothing ini it!

In any case, references to other Make variables in the *value* of a Make variable declaration are not expanded until they are actually needed. In other words you cannot work out the eventual

result of a variable expansion from the order in which things are declared. At first glance, it looks as though the command in the 'm4' target in *example 5.13* will execute gcc:

```
LD       = gcc $(CFLAGS)
LDFLAGS  = -static
LINK     = $(LD) $(LDFLAGS)
...
m4: $(m4_OBJECTS)
        $(LINK) -o m4 $(m4_OBJECTS) $(m4_LDADD) $(LIBS)

LD = ld
```

Example 5.13: *Make variables are not expanded until they are used*

Not so. Until the moment the value of '$(LINK)' is needed in the command associated with the 'm4' target, the values of '$(LD)' and '$(LDFLAGS)' are not expanded. By the time this happens, '$(LD)' contains a different value to what it held when 'LINK' was declared:

```
$ make m4
ld -static -o m4 main.o freeze.o stackovf.o ../m4/libm4.a
```

So, as Make expands the variables in the command before passing them to the shell for execution, '$(LINK)' unrolls like this:

```
$(LINK)
↦ $(LD) $(LDFLAGS)
⇒ ld -static
```

As a consequence, there can be no loops in variable references, even indirectly, otherwise Make would get stuck in an infinite recursion loop. GNU Make is able to detect such loops, and will diagnose them for you. For example, here is a 'Makefile' with a reference loop:

```
FOO = foo $(BAR)
BAR = $(FOO) bar

all:
        echo $(FOO)
```

Example 5.14: *A Makefile with a variable reference loop*

When asked to refresh the target, GNU Make bails out with the following error message:

```
$ make
Makefile:4: *** Recursive variable 'FOO' references
Makefile:4:     itself (eventually).  Stop.
```

While preparing the *command* (at line 4 of the 'Makefile') for processiong, Make will try to expand variable references, and notice that there are still unexpanded variables with a *variable-name* that has already been expanded:

```
echo $(FOO)
↦ echo foo $(BAR)
↦ echo foo $(FOO) bar
```

Usually, it is very easy to remove the circular reference once GNU Make has given you the *variable-name* and 'Makefile' line number where it detected the loop.

5.2.3 Make Comments

A comment consists of the text starting with a '`#`' sign and continuing through to the end of the line. Make will understand a comment anywhere within a '`Makefile`', except within the *command* part of a rule. Although you can put a comment inside a multi-line variable assignment, it is generally a bad idea because it is hard to tell whether the following line is still part of the comment or not[4].

```
libm4_a_SOURCES         = builtin.c debug.c error.c eval.c hash.c \
                          # Do the next 2 lines continue this comment? \
                          input.c ltdl.c macro.c module.c output.c \
                          path.c regex.c symtab.c utility.c
```

Example 5.15: *Ambiguous comment endings are bad*

5.3 Shell Commands

You have probably noticed that the *command* part of the rules shown so far are just executing programs that are also available to users from the command line. In fact, each line of *command* is evaluated individually. When you write *command*s that use shell syntax, each line in the *command* part of the rule is individually executed in its own shell – unless you tie consecutive lines together with backslashes.

```
includedir      = /usr/local/include
include_HEADERS = m4module.h error.h hash.h system.h
...

install-HEADERS: $(include_HEADERS)
        mkdir $(includedir)
        for p in $(include_HEADERS); do \
          cp $$p $(includedir)/$$p; \
        done
```

Example 5.16: *Basic header installation Makefile excerpt*

In this example, the *command* consists of only two actual shell commands. The first, '`mkdir $(includedir)`', is invoked as a simple one line *command*; the next *command* consists of the remaining lines in the rule, because of the '`\`' at the end of each unfinished line.

There is an important distinction to be made here between *Make* variables, like '`$(includedir)`', and *shell* variables, such as '`$$p`'. We have already discussed Make variables in Section 5.2.2 [Make Variables], page 109. Unfortunately the '`$`' symbol is already used to denote Make variables, and consequently to pass '`$`' through to the shell from the *command* part of a rule the '`$`' symbol needs to be doubled up: Hence the '`$$p`' in this *command* becomes '`$p`' when finally executed by the shell.

Describing how to program in shell is beyond the scope of this book, but you can find terse details in your system manual pages:

```
$ man sh
```

See Section 5.8 [Further Reading], page 125 for further recommendations. In practice, provided that you use Automake in conjunction with Make, it is quite unusual to need shell programming features in your Makefiles, since all of the complicated commands are generated for you. Automake is dicussed in much more detail later, in Chapter 10 [Automake], page 203.

[4] Actually, with GNU Make, it *is* still part of the comment.

5.3.1 Command Prefixes

This is what happens when the '`install-HEADERS`' target from *example 5.16* is invoked:

```
$ make install-HEADERS
mkdir /usr/local/include
mkdir: cannot make directory ‘/usr/local/include’: File exists
make: *** [install-HEADERS] Error 1
```

Not exactly what we wanted, yet we do need to keep the '`mkdir`' command line in the rule incase the person who installs the project really doesn't have a '`/usr/local/include`' directory. Discarding the error message is easy enough, we simply redirect it to '`/dev/null`'. The real problem is that when Make encounters an error in the *command* sequence, it stops processing and prints a message like the one above.

Make provides a facility to override this behaviour, and thus continue normal processing and ignore any error from the shell command. By prefixing the command with a '`-`' character as follows, we tell `make` that it doesn't matter if that command fails:

```
install-HEADERS: $(include_HEADERS)
        -mkdir $(includedir) 2>/dev/null
        for p in $(include_HEADERS); do \
          cp $$p $(includedir)/$$p; \
        done
```

Example 5.17: *Ignoring errors from shell commands in Make rules*

Things progress as expected now:

```
$ make install-HEADERS
mkdir /usr/local/include
make: [install-HEADERS] Error 1 (ignored)
for p in m4module.h error.h hash.h system.h; do \
  cp $p /usr/local/include/$p; \
done
```

Notice that the '`$$p`' references from the Makefile commands have been passed to the shell as '`$p`' as explained in the last section.

As long as it doesn't encounter any errors in the *command*s as it processes them (except for '`-`' prefix ignored errors), the normal behaviour of Make is to echo each line of the command to your display just before passing it to the shell. As this happens, Make variables are replaced by their current value, as evidenced by the list of files in the `for` loop above where '`$(include_HEADERS)`' was written in the Makefile. Although double '`$$`' sequences are replaced by a single '`$`', the command itself along with any shell variable references are then simply copied to your display before the shell processes it.

There is another prefix that you can add to a command line to suppress echoing. By inserting a '`@`' at the start of a command, you tell Make to pass that command directly to the shell for processing. We can take advantage of this feature to clarify what the shell is doing in the *command*s of our '`install-HEADERS`' rule:

```
install-HEADERS: $(include_HEADERS)
        @test -d $(includedir) || \
          { echo mkdir $(includedir); mkdir $(includedir); }
        @for p in $(include_HEADERS); do \
          echo cp $$p $(includedir)/$$p; \
          cp $$p $(includedir)/$$p; \
        done
```

Example 5.18: *Suppressing echoing of shell commands in Make rules*

There are still two commands in this rule, but now we have turned off echoing from Make with the '`@`' prefix, and instead use the shell `echo` command to report what is being done:

```
$ make install-HEADERS
cp m4module.h /usr/local/include/m4module.h
cp error.h /usr/local/include/error.h
cp hash.h /usr/local/include/hash.h
cp system.h /usr/local/include/system.h
```

This time, we tested for the existence of a '`$(includedir)`', and created it only if it was missing. In this example the directory was already present, so the second clause of the first command in *example 5.18* didn't trigger.

Because of the '`@`' prefix on the first command in the rule, no command is echoed by make. We control when something is displayed, and echo the command only when it is executed. Much better.

Also, rather than letting Make echo the `for` loop code directly, we have the shell individually report each file that it copies, which makes the output of the command reflect exactly what is going on, and makes it easier for your users to see what is happening when that rule is invoked.

5.4 Special Targets

Different implementations of Make provide varying numbers of *special targets*: GNU Make itself has a particularly impressive array of special targets, which are described exhaustively in the documention installed along with the GNU Make binary. However, in conjunction with Automake (see Chapter 10 [Automake], page 203), none are actually explicitly required. We discuss them here because if you decide not to use Automake for some reason, you *will* need to use them – and, besides, the equivalent features in Automake are easier to understand if you have a good grasp of what is happening in the '`Makefile`'s it generates.

In the next two subsections we discuss '`.SUFFIXES`' and '`.PHONY`' as a useful and representative subset of the special targets supported by all incarnations of Make.

5.4.1 Suffix Rules

The Makefile in *example 5.11* has a dependency rule for the compilation of each of the source files in the source directory. Remembering to add a new rule to the Makefile each time a new source file is added to the project would be an unwanted distraction from the real work of writing code. Accordingly, Make offers a way to specify how groups of similar files are kept up to date, which is especially useful in cases like this where the commands for each rule are almost identical.

Make will determine when to apply these rules based on the file suffix that follows the last dot in the pertinent file name. The explicit rules all follow this template:

```
main.o: main.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c main.c
```

Example 5.19: *A Make rule for compiling main.c*

The relationship between the target, '`main.o`', and its dependency, '`main.c`' is characterised by the suffixes '`.o`' and '`.c`'. An equivalent suffix rule, which can also be used to update any target with a '`.o`' suffix from an existing '`.c`' suffixed filename is as follows:

```
.c.o:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $<
```

Example 5.20: *A suffix rule for compiling C source files*

Confusingly, if you compare *example 5.19* and *example 5.20*, the order of the suffixes to the left of the '`:`' above, is reversed with respect to how they would appear in the earlier explicit rule. This takes some getting used to, but follows the precedent set by prehistoric implementations of the Make utility. The odd looking variable reference, '`$<`' refers to the '`.c`' file that matched the rule, and is explained fully in Section 5.4.2 [Automatic Variables], page 117.

Suffix rules take a very similar form to normal dependency rules. However, the *target-suffix* follows the *dependent-suffix* immediately with no separation:

```
dependent-suffixtarget-suffix:
        command
        ...
```

Example 5.21: *Format of a double suffix rule*

Notice that unlike *example 5.3* there is nowhere to list *dependencies* in a suffix rule. As a matter of fact, Make will not interpret a rule that *does* list *dependencies* as a suffix rule; it will be treated as a dependency rule with a *target* named, say, '`.c.o`'. GNU Make already knows about many of the common suffixes like '`.c`' and '`.o`', and in fact has a fairly exhaustive list of suffixes and default suffix rules declared for you, as explained in Section 5.7 [Invoking Make], page 123. It uses this knowledge to distinguish between a suffix rule for generating '`.o`' files from similarly named '`.c`' files, and a standard dependency rule to create a wierd target file named '`.c.o`'. You **must** declare any additional suffixes used by the suffix rules in your '`Makefile`'. Make uses a special target named *.SUFFIXES* for this, like so:

```
.SUFFIXES: .c .o
```

The default suffix list is a double edged sword though, and will occasionally elicit surprising behaviour from Make when some of your files happen to match one of the predeclared suffix rules. We advocate the *principle of least surprise*[5], and as such recommend enabling only the suffix rules that are specifically required by each individual '`Makefile`'. First, empty the suffix list with an empty '`.SUFFIXES`' target, and then declare the suffixes you actually need in a second '`.SUFFIXES`' target.

This works because, unlike normal targets, when '`.SUFFIXES`' appears multiple times within a '`Makefile`', the dependencies for each new appearance are added to a growing list of known file suffixes. Except that at any point '`.SUFFIXES`' is written with no dependencies, this growing list is reset by removing every suffix it contains – subsequent appearances will then be able to add more suffixes back in to the list.

[5] Complex systems are much easier to understand if they always do what you would expect.

Applying these idioms to our Makefile, we now have:

```
CC       = gcc
CFLAGS   = -ggdb3
CPPFLAGS = -DDEBUG=1

m4_OBJECTS = main.o freeze.o stackovf.o
m4_LIBS    = ../lib/libm4.a

.SUFFIXES:
.SUFFIXES: .c .o

m4: $(m4_OBJECTS) $(m4_LIBS)
        $(CC) $(CFLAGS) -o m4 $(m4_OBJECTS) $(m4_LDADD) $(LIBS)


.c.o:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $<
```

Example 5.22: *The complete Makefile with suffix rule*

In general, you will need a mixture of suffix rules and target dependency rules to describe the entire build process required to transform your source code into an executable. Sometimes you might want to use a single suffix rule for the vast majority of your files, but make an exception for an odd one or two.

For example, one set '`CPPFLAGS`' might be fine for most of your C compilation, but you need to add an additional '`-DDEBUG=1`' option to one or two special files:

```
CC       = gcc
CFLAGS   = -ggdb3
CPPFLAGS =

...

.c.o:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $<


stackovf.o: stackovf.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -DDEBUG=1 -c stackovf.c
```

Example 5.23: *Compiling some files with different options*

Here, the files compiled by the suffix rule use only the options specified in the variables, but '`stackovf.o`' is compiled by a specific rule which explicitly adds '`-DDEBUG=1`'. By listing the file that needs different options as a specific target, that rule overrides the suffix rule in that case.

Let's see this in action:

```
$ rm *.o
$ make m4
gcc -ggdb3  -c main.c
gcc -ggdb3  -c freeze.c
gcc -ggdb3  -DDEBUG=1 -c stackovf.c.c
gcc -o m4 main.o freeze.o stackovf.o ../lib/libm4.a
```

5.4.2 Automatic Variables

Automatic variables are used in much the same way as normal Make variables, except that their value are set automatically by Make according to the suffix rule in which they are used.

In *example 5.20*, we used a so called *automatic variable* to refer to the implicit *dependency* file in a suffix rule. The corollary of this is an automatic variable to refer to the implicit *target* in a suffix rule. Here is a variation of the suffix rule we added to our '`Makefile`' in *example 5.20*, but using automatic variables to refer to the implicit *target* and *dependency* files:

```
.c.o:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $@
```

Example 5.24: *Suffix rule to compile C source files*

'`$<`' In a suffix rule, this variable contains the name of the dependent file that would be needed for the rule to match.

When Make determines that it needs an up to date '`stackovf.o`' in order to continue the build, and decides to use this suffix rule to bring it up to date, '`$<`' will expand to '`stackovf.c`'.

'`$@`' Similarly, this variable contains the name of the target file which matched this suffix rule.

For the same '`stackovf.o`' problem above, '`$@`' will expand to '`stackovf.o`'.

A small example should crystalise these facts in your mind. Using the following tiny '`Makefile`' you can see how this works in practice:

```
.c.o:
        @echo '$$< = $<'
        @echo '$$@ = $@'
```

Example 5.25: *Automatic variable setting demonstration*

You can now create any dummy '`.c`' file to match the dependency part of the suffix rule, and ask make to bring a matching target up to date:

```
$ touch hello.c
$ make hello.o
$< = hello.c
$@ = hello.o
```

NOTE GNU Make will actually honour automatic variables in normal dependency rules too, but this is an enhancement beyond what many other implementations of Make allow. If you use this feature in your own project, you can be certain that one of your users will be using a `make` that doesn't support it, and will be unable to compile your code as a result.

5.4.3 Phony Targets

Often, for ease of use, you want to capture a useful set of shell commands in a '`Makefile`', yet the commands do not generate a file and thus have no *target* file; the *target* is just a convenient

handle for a command that you want to run periodically. The '`clean`' target from *example 5.2* is an example of such a rule, repeated here:

```
clean:
        rm -f $(m4_OBJECTS)
```

Example 5.26: *A phony target*

There is no file '`clean`' involved here; but you *can* clean out the object files in the build directory with:

```
$ make clean
rm -f main.o freeze.o stackovf.o
```

There is nothing special happening here. Asking Make to build the '`clean`' target, it finds that the '`clean`' file is out of date (missing entirely, infact!) and tries to refresh it by running the *command*s in the associated rule. No '`clean`' file is created by running *command*s, which means that Make will consider the '`clean`' target to still be out of date the next time you invoke `make clean`.

So what happens if you accidentally drop a file named '`clean`' into the build directory, and try to invoke the '`clean`' target? Well, Make will see the file, and find that there are no *dependencies* against the '`clean`' target – so '`clean`' is up to date already, and there is no need to execute the `rm` command. In all fairness, if you choose the names of your phony targets carefully, you shouldn't get into this situation. Just the same, Make provides a special target, *.PHONY*, against which you can list your phony targets as *dependencies*, like this:

```
.PHONY: clean
```

Make now knows that '`clean`' does not represent an actual file, and will always execute the *command* part of the '`clean`' rule.

Unlike normal targets, '`.PHONY`' can be inserted into a Makefile more than once, in which case Make will append subsequent '`.PHONY`' *dependencies* to the current list, in exactly the same way that '`.SUFFIXES`' behaves (see Section 5.4.1 [Suffix Rules], page 114).

5.5 Make Conditionals

GNU Make provides a rich set of conditional directives, with which you can annotate parts of the '`Makefile`' to be used conditionally (or ignored). Make conditional directives work in very much the same way as C preprocessor directives ((**FIXME:** *xref the preprocessor section of the GCC chapter.*)); before the effective contents of a Makefile are parsed, the file is examined and the conditional directives processed to determine what the effective contents of the Makefile are. There are four Make conditional directives, all of which take the following form:

```
conditional-directive
        effective if condition is true
        ...
else
        effective if condition is false
        ...
endif
```

Example 5.27: *Format of a Make conditional directive*

The '`else`' keyword, and the following *false* text can be omitted entirely if necessary. There is no need to start the directives in column zero, nor to indent the conditional text blocks any

differently than normal. Whatever indentation is given for conditionl text is used verbatim in the effective contents of the '`Makefile`' after the directives have been parsed. Should you wish to indent the directives themselves, you must use spaces rather than tabs to avoid fooling Make into thinking the directive is part of a dependency rule *command*.

The *conditional-directive* can be one of the following:

`ifdef` *`variable-name`*

The *if defined* directive tests whether the named Make variable (see Section 5.2.2 [Make Variables], page 109) is empty. However, *variable-name* is not actually expanded: If its value contains anything, even a reference to some undefined Make variable, the '`effective if condition is true`' text is passed through to the next stage of the Makefile parser. Converseely, if an else clause is present and *variable-name* has an empty value, or is not defined at all, the condition fails and the '`effective if condition is false`' text is effective.

```
OPT        =
CFLAGS     =
ALL_CFLAGS = $(OPT) $(CFLAGS)
all:
ifdef ALL_CFLAGS
        @echo true
else
        @echo false
endif
```

Here, the '`Makefile`' above will echo '`true`'. Although '`ALL_CFLAGS`' *evaluates* to empty, the pre-processing pass merely checks to see if the value of '`ALL_CFLAGS`' *contains any text* – which it does – so the '`@echo true`' text is effective.

`ifndef` *`variable-name`*

The *if not defined* conditional works in the opposite sense to '`ifdef`'. By implication, if *variable-name* has an empty value, or is simply not defined, the '`effective if condition is true`' text is effective, otherwise, if an else clause is present, the '`effective if condition is false`' text is effective.

```
CC =
ifndef CC
CC = gcc
endif

all:
        @echo $(CC)
```

When parsed by Make, the '`Makefile`' above will always '`echo gcc`'.

`ifeq "`*`argument`*`" "`*`argument`*`"`

Unlike the last two conditionals, this *if equal* directive *does* expand any variables referenced in either of the two *argument*s, before they are compared to determine the effective text. If both *argument*s have the same contents after variable references have been expanded, then the '`effective if condition is true`' text is effective.

```
ifeq "$(CC)" "gcc"
CFLAGS = -Wall -pedantic
endif
```

This example shows how you might change the '`CFLAGS`' to give better warnings when compiling with `gcc`.

`ifneq "`*argument*`" "`*argument*`"`

Similarly, there is an *if not equal* directive, which behaves much the same as '`ifeq`', but obviously with the sense of the test reversed.

```
ifneq "$(CC)" "gcc"
CFLAGS=-g
else
CFLAGS=-ggdb3
endif
```

This '`Makefile`' fragment shows how you can set the compiler debugging flags optimally for GCC, without adverse effects when a different compiler is used.

5.5.1 Make Include Directive

GNU Make has one other directive, which is not actually a conditional, though it is processed in the same pass as the conditionals and has some noteworthy interactions with them:

`include` *file-name* `...`

Example 5.28: *Format of a Make include directive*

Again, in parallel with the C preprocessor, before the effective contents of the Makefile are parsed, Make's '`include`' directive is effectively replaced by the listed *file-name* contents. Any variable references in the list are expanded before the filenames are opened, and since the names are delimited with whitespace, a single referenced variable can safely contain more than one *file-name*. This does mean, however, that there is no way to specify a *file-name* containing a space.

If *file-name* is a relative path, such as '`../vars.mk`', instead of a fully qualified path like '`/usr/share/make/make.std`', and cannot be found by starting in the current directory, Make tries to find the file in directories in the *include search path*. See Section 5.7 [Invoking Make], page 123, for details about setting the include search path. If a *file-name* still cannot be found, then Make will issue a warning:

```
Makefile:1: Make.depend: No such file or directory
```

If any included *file-name* was not found, but has a corresponding *target* rule, Make will issue these warnings, but then try to build the missing files and continue. This is easiest explained by enhancing our '`Makefile`'. As with any non-trivial project, there are many interdependencies between the various source files that comprise the GNU M4 package. In the '`Makefile`' we have been looking at so far in this chapter, each of the '`m4_OBJECTS`' files depends on parts of '`../lib/libm4.a`'. In fact, it is important that the objects be refreshed if they become out of date with respect to those parts they depend on...

Maintaining the dependencies between files is time consuming and error prone. UNIX comes with a tool called `makedepend`, which will automatically discover the dependencies for us, and we can put it to good use here:

```
MAKEDEPEND     = makedepend
DEPEND_FRAGMENT= Make.depend

include $(DEPEND_FRAGMENT)

CC       = gcc
CFLAGS   = -ggdb3
CPPFLAGS = -DDEBUG=1
```

```
m4_SOURCES = main.c freeze.c stackovf.c
m4_OBJECTS = main.o freeze.o stackovf.o
m4_LIBS    = ../lib/libm4.a

.SUFFIXES:
.SUFFIXES: .c .o

m4: $(m4_OBJECTS) $(m4_LIBS)
        $(CC) $(CFLAGS) -o m4 $(m4_OBJECTS) $(m4_LDADD) $(LIBS)


.c.o:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $<

$(DEPEND_FRAGMENT): $(m4_SOURCES)
        $(MAKEDEPEND) $(CPPFLAGS) -f $(DEPEND_FRAGMENT) $(m4_SOURCES)
```

Example 5.29: *Automatic file dependencies with Make*

Now, we are trying to '`include`' a Makefile fragment which contains the generated dependency list. But rather than manually generating it, we have also added a new rule which uses the `makedepend` utility to write the dependencies to '`Make.depend`'. Now, even though Make initially warns us if it cannot '`include`' '`Make.depend`', before giving up it will try to build the file itself, and then reread the original '`Makefile`'.

Notice that we have made the depend-fragment file dependent upon the source files it scans, so that if they change, '`Make.depend`' will be refreshed. The first time '`Makefile`' is used after adding these new items, when '`Make.depend`' does not yet exist, you will see the following:

```
$ make
Makefile:4: Make.depend: No such file or directory
makedepend -DDEBUG=1 -f Make.depend main.c freeze.c stackovf.c
cp: Make.depend: No such file or directory
Appending dependencies to Make.depend
make: 'm4' is up to date.
```

Make eventually created its own '`Make.depend`', and now knows how to regenerate it if any of the sources whose dependencies it contains are changed!

You will be able to find more details about the `makedepend` command in your system manual pages, with *man makedepend*.

5.6 Multiple Directories

Except in the most trivial of projects, source files are usually arranged in subdirectories. Unfortunately, because of the way it is designed, Make is difficult to use with files that are not in the same directory as the Makefile itself. Because of this, it is best to put a separate Makefile in each directory, and make each Makefile responsible for only the source files in its own directory. For example:

```
$ tree -P Makefile m4
  m4
  |-- Makefile
  |-- config
  |   `-- Makefile
  |-- doc
  |   `-- Makefile
  |-- examples
  |   `-- Makefile
  |-- intl
  |   `-- Makefile
  |-- m4
  |   `-- Makefile
  |-- modules
  |   `-- Makefile
  |-- po
  |   `-- Makefile
  |-- src
  |   `-- Makefile
  `-- tests
      `-- Makefile

  10 directories
```

In order to have the build process recurse through the source tree, the toplevel Makefile must descend into the subdirectories and start a new Make process in each. Typically, the order that the subdirectories are visited is critical. For the project depicted above, the build must first visit the '`po`' and '`intl`' directories to build the internationalisation libraries[6], then the '`m4`' subdirectory to build '`libm4.a`', which uses the internationalisation library, before building the loadable modules which rely on '`libm4.la`' and so on, and so forth, until the build completes by building the documentation in the '`doc`' subdirectory.

GNU Make sets the variable '`$(MAKE)`' to the name with which `make` was invoked. For example, many vendor environments install GNU Make as '`gmake`' – blindy running `make` from the *command* part of a rule would not then invoke `gmake`, even if the user had invoked `gmake` on the top level Makefile.

```
SUBDIRS = intl po config m4 modules src tests examples doc

all:
        @for subdir in $(SUBDIRS); do \
          echo "Making all in $$subdir"; \
          cd $$subdir && $(MAKE) all; \
        done
```

Example 5.30: *Top level Makefile fragment for recursing subdirectories*

Using '`$(MAKE)`' in *example 5.30* ensures that the recursive `make` invocations in the *command* will execute the same program that the user originally ran to start the instance of `make` that read this Makefile.

With care, it is possible to set up a Makefile that, in addition to running recursive `make` invocations on Makefiles in subdirectories, will also perform some builds in its own directory.

[6] We already covered this in (**FIXME:** *xref chapter 2.*).

And there is certainly nothing to prevent you from setting up a build tree that has more than nested levels of Makefile. Later in Chapter 10 [Automake], page 203, we will explain how Automake manages all of the details of recursion for you.

5.7 Invoking Make

The majority of the time, with the hard work already expended in creating a '`Makefile`', employing that file to refresh build targets is no more complicated than this:

```
$ make target
```

This being the GNU system though, you can of course do so much more. There are many command line options that affect the way that GNU Make behaves, which you can always get a summary of by running `make --help` from the command line. We will describe a useful subset in this section, but you can find comprehensive details in *The Gnu Make Manual* that ships with GNU Make.

'`-I directory`'
: You can specify this option as many times as you like to list various directories that you want Make to search for additional Makefile fragments that are pulled in to the effective Makefile with the '`include`' directive.

'`-f file-name`'
: This option allows you to specify an alternative Makefile by name. Instead of reading from the default '`Makefile`', *file-name* will be read instead.

'`-n`'
: If you need to see what Make *would* do, without actually running any of the command rules, use the '`-n`' option to have Make display those commands but not execute them.

'`-k`'
: Occasionally, you might be missing one of the tools that a shell command early in the sequence tries to run, but which is not critical to the correct operation of the package, but even so Make will give up and report an error message. This option tells Make to just keep going in the face of errors from shell commands, so that you can snatch victory from the jaws of defeat.

One of the most useful facilities is being able to override Make variable values from the command line: With this feature you can set your Makefile variable to perform a production build of your project (say, maximum optimisation, create shared libraries, minimum debug code enabled), but specify more appropriate values on the command line during development. So, in our '`Makefile`' we would specify the following:

```
CFLAGS   = -O6
CPPFLAGS =
```

Example 5.31: *Production build values for compilation flags*

But during development, by invoking Make as follows, change the values of those Make variables for the current build:

```
$ make CFLAGS='-ggdb3 -Wall' CPPFLAGS='-DDEBUG=9' m4
```

Do be aware that it is easy to end up with a set of mismatched objects, compiled with different flags if you keep changing the override values without performing a complete build. To avoid doing this accidentally, you should probably run '`make clean`' between changes to the override settings. Unless you really do want to compile just a few objects with different build options for some reason...

5.7.1 Environment Variables

In addition to having a provision for overriding the values of Make variables at build time, Make also provides for supplying default values for variables that are not otherwise assigned. The mechanism for enabling this feature is simply through the UNIX environment. If you are using a bourne compatible shell[7], variables are exported to the environment using the `export` keyword, like this:

```
$ prefix=$HOME/test
$ export prefix
```

With the '`prefix`' shell variable exported into the environment, it can be referred to from a Makefile with '`$(prefix)`', just like any other Make variable. However, it is only a **default** value, so it won't be seen if there is a declaration of '`prefix`' within the '`Makefile`', or indeed if you override it from the command line as described in the last section.

GNU Make will allow you to force the Makefile to prefer settings from the environment over the variable declarations in the file if you specify the '`-e`' option when you invoke `make`. None of this is as complicated as it sounds – take the following '`Makefile`' fragment:

```
includedir      = $(prefix)/include
include_HEADERS = m4module.h error.h hash.h system.h
...
install-HEADERS: $(include_HEADERS)
        @test -d $(includedir) || \
          { echo mkdir $(includedir); mkdir $(includedir); }
        @for p in $(include_HEADERS); do \
          echo cp $$p $(includedir)/$$p; \
          cp $$p $(includedir)/$$p; \
        done
```

Example 5.32: *Precedence of Make variable declarations*

For this particular '`Makefile`' we can *override* the value of '`includedir`' like this:

```
$ make -n includedir='$(HOME)/include'
mkdir /home/gary/include
cp m4module.h /home/gary/include/m4module.h
...
```

Notice that I set the override value of '`includedir`' to reference the variable '`$(HOME)`', and this *was* defaulted from my shell environment, since there is no declaration for '`HOME`' either in the '`Makefile`' or in the command line invocation.

However, there would be no point in setting '`includedir`' in the shell as an environment variable, since whatever default value was set, it would be superceded by the variable delaration at line 1 of '`Makefile`'. Unless, we use the '`-e`' option to `make`:

```
$ prefix=/usr/local
$ export prefix
$ make -n -e
cp m4module.h /usr/local/include/m4module.h
cp error.h /usr/local/include/error.h
...
```

[7] For example GNU bash, ksh, zsh, sh5 are all based on Steve Bourne's original UNIX shell.

5.8 Further Reading

Once you have read and understood this chapter, you will have all the information you need to get the most from the following chapters, especially Chapter 10 [Automake], page 203 which builds directly on the material discussed here. In the short space allocated to Make in this book, we have only really covered the basics of the functionality and application of the UNIX Make tool, and barely scratched the surface of the many extensions provided by GNU Make. If you wish to find out more about Make, or the shell language used in the *command* parts of dependency rules, here are some other books we recommend:

The GNU Make Manual
: Written by The Free Software Foundation
Available with the sources for GNU Make

Managing Projects with make
: Written by Andrew Oram and Steve Talbot
Published by O'Reilly; ISBN: 0937175900

Learning the Bash Shell
: Written by Cameron Newham and Bill Rosenblatt
Published by O'Reilly; ISBN: 1565923472

The Bourne Shell Quick Reference Guide
: Written by Anatole Olczak
Published by ASP; ISBN: 093573922X

Before we revisit the application of Make, in the next few chapters we will discuss some of the other tools that underpin Automake, namely Autoconf and M4. But first we will return to the discourse on compiler tools from the end of the last chapter on GCC. If you are more interested in learning about the GNU configuration tools, you could skip straight to the chapter about M4, See Chapter 8 [Writing M4 Scripts], page 187.

6 Scanning with Gperf and Flex

6.1 Scanning with Gperf

A part of the job of scanners is recognizing keywords amongst identifiers. We examine this task briefly in Section 6.1.1 [Looking for Keywords], page 127. This will lead to us to design a program automating the generation of keyword recognizers, which turns out to be what Gperf is, as explained in Section 6.1.2 [What Gperf is], page 130.

Once the generic background is depicted, we will proceed with an example presenting the most basic features of Gperf, see Section 6.1.3 [Simple Uses of Gperf], page 131. Then, after having presented more formally `gperf` in Section 6.1.4 [Using Gperf], page 133, we will present a complete use of Gperf, exhibiting the classic pitfalls, Section 6.1.5 [Advanced Use of Gperf], page 135, and its interface with Autoconf and Automake, Section 6.1.6 [Using Gperf with the GNU Build System], page 140.

Finally, in Section 6.1.7 [Exercises on Gperf], page 141, we will propose a few directions for the readers willing to go further with Gperf.

6.1.1 Looking for Keywords

Suppose you face the following problem: you are given a fixed and finite set K of k words —henceforth *keywords*—, find whether a candidate *word* belongs to K.

There are tons of solutions, of varying efficiency. The question essentially amounts to writing a generalization of `switch` operating on literal strings:

```
switch (word)
{
  case "foo":
    /* perform 'foo' operations. */
    break;
  case "bar":
    /* perform 'bar' operations. */
    break;
  ...
  default:
    /* WORD is not a keyword. */
    break;
}
```

A first implementation could be

```
if (!strcmp (word, "foo"))
  /* perform 'foo' operations. */
else if (!strcmp (word, "bar"))
  /* perform 'bar' operations. */
  ...
else
  /* WORD is not a keyword. */
```

which of course is extremely inefficient both when submitted a keyword (think of the last one) and worse yet, when submitted a plain word. In the worst case, we perform k string comparisons. But it is good enough for a few lookups in a small set of words, for instance command line arguments.

The proper ways to implement this by hand are well known, binary search for a start:

```
/* Compare the keys of KEY1 and KEY2 with strcmp. */
int keyword_cmp (const keyword_t *key1, const keyword_t *key2);

/* The list of keywords, ordered lexicographically. */
const keyword_t keywords[] =
{
  ...
  { "bar", &bar_function },
  { "baz", &baz_function },
  ...
}

/* Number of keywords. */
const size_t keywords_num = sizeof (keywords) / sizeof (*keywords);

{
  /* Look for WORD in the list of keywords. */
  keyword = bsearch (word, keywords, keyword_num,
                     sizeof (keyword_t), keyword_cmp);
  if (keyword)
    /* Perform the associated action. */
    keyword->function ();
  else
    /* Not a keyword. */
    default_action ();
  ...
```

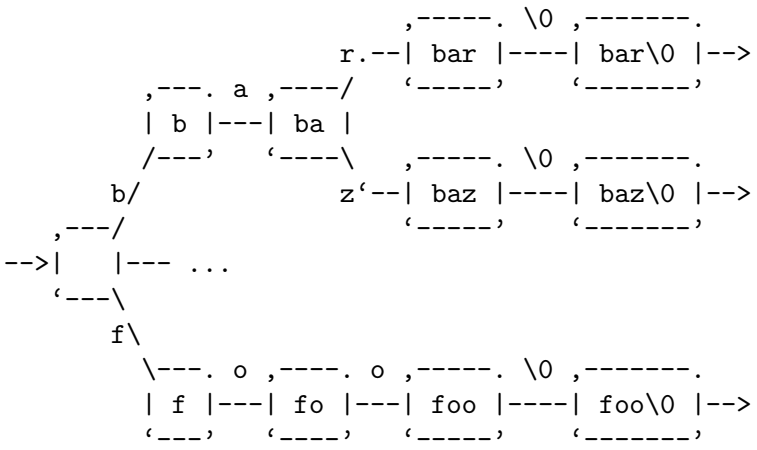
This is very efficient —the number of string comparisons is bounded by the logarithm of k— and way enough for most uses. Nevertheless, you have to be *sure* that your array is properly sorted. For sake of reliability, sorting the array beforehand (e.g., using `qsort` at program startup) might be a good idea, but of course incurs some performance penalty.

But initialization is unlikely to matter, especially if keyword lookup is frequently performed. If you are crazy for speed, always looking for wilder sensations, you will notice that once we reached some performance bounds with respect the number of *string* comparisons, then it is the number of *character* comparisons that matters. Imagine you have 256 keywords, then the first character of a non keyword will be compared at least 8 times, no matter whether any keyword actually starts with this letter. All its other characters are also likely to be read and compared several times. Therefore, instead of considering the problem looking at rows of strings, you will look at the columns of characters. You will end up with something like:

```
switch (word[0])
{
  ...
  case 'b':                          /* 'b' */
    switch (word[1])
    {
       ...
       case 'a':                     /* 'ba' */
         switch (word[2])
         {
            ...
            case 'r':                /* 'bar' */
              switch (word[3])
              {
                case '\0':           /* 'bar\0' */
                  /* Perform 'bar' function. */
                  break;
                ...
                default:
                  /* Not a keyword. */
                  break;
              } /* switch (word[3]) */
              break;
            ...

         } /* switch (word[2]) */
         break;
       ...
    } /* switch (word[1]) */
    break;
  ...
} /* switch (word[0]) */
```

In other words, you will implement by hand a small automaton, reduced to a simple tree:



Example 6.1: *A Fast Keyword Tree-like Recognizer*

where (i) the entering arrow denotes the initial state, (ii) exiting arrows denote successful keyword recognitions, and (iii), any characters that cannot be "accepted" by a state (i.e., there is no

transition, no arrow, exiting from this node, labeled with the character), result in the successful recognition of a *non* keyword.

This automaton is definitely unbeatable at recognizing keywords: how could you be faster given that each character is compared only once[1]! The average time taken by linear approach, first exposed, is proportional to the number of keywords; using a binary search cuts it down to its logarithm; and finally the automaton needs virtually a single string comparison: its efficiency is independent of the number of keywords!

Can you do better than that?...

In typical compilers input, you have to take user identifiers into account, and on average, being the fastest at recognizing keywords is not enough: you would like to discriminate non keywords as soon as possible. Suppose you found out that all the keywords end with '`_t`', or even that the same character always appears at some fixed position. Starting by checking the characters at these positions will let you recognize some non keywords faster than using the automaton above.

Generalizing this idea, we are looking for a function which will let us know with a high probability whether a *word* is not a keyword just by looking at a few well chosen characters. We can improve even further this idea: if we choose the characters well enough, it might give us an intuition of the keyword the *word might* be. Let us name this function `hash`, and let it compute an integer, so that processing its result will be cheap. The algorithm will be:

```
compute the hash code: the value of 'hash (word)'

if the hash code is not representative of any keyword then
  return failure
else
  for each keyword corresponding to the hash code do
    if this keyword matches then
      return it
    end if
  end for each
  return failure
end if
```

This algorithm, based on a hash function, is as efficient as the automaton was: its performances are independent of the number of keywords!

It is crucial that the hash code be simple to compute, but the time spent in *designing* hash function is inessential: once you wrote it for a given set of keywords, it will be potentially used millions of times. But of course a little of automation would be most welcome, since laziness is definitely a quality of programmers...

6.1.2 What Gperf is

Gperf is a generator of small and fast recognizers for compile-time fixed sets of keywords. Unless specified differently through command line options, the result is C code consisting of a static hash table and a hash function optimized for a given set of keywords. There is no restriction on the use of its output, v.g., no license is imposed on its output.

[1] There is room for debate here: the compiler will transform most of the `switch` into plain `if`s, instead of computed `goto`s, especially when there are few `case`s, hence there will be several comparisons. A strict study depends upon the compiler. In addition, the price of the `goto`s should be taken into account too. Let us merely say "each character is consulted once".

This recognizer is typically used to discriminate reserved words from identifiers in compilers. The GNU Compiler Collection (**FIXME:** *ref to GCC.*), GCC, uses it at least for Ada, C, C++, Chill, Java, Modula 2, Modula 3, Objective C, and Pascal. GNU Indent also bases its keyword recognition on Gperf, but its uses are not limited to programming languages. For instance `makeinfo`, the Texinfo to Info translator, uses it to recognize its directives. Finally, it proves itself useful for handling sets of options, especially because it provides a very convenient interface —performance is less likely to matter. For instance GNU a2ps uses Gperf to read its configuration files, GNU Libiconv, the GNU character set conversion library, uses Gperf to resolve more than three hundred aliases such as 11 different names for ASCII.

A *hash function* is a fast function which maps any member of a k element user-specified keyword set K onto an integer range 0..n - 1. The result integer, called *hash code*, is then used as an index into an n element table containing the keywords, sorted according to their hash code.

A hash function is *perfect* if no two keywords have the same hash code, in other words if the result hash table is collision-free. This means that at runtime time cost is reduced to the minimum: examining a single string at runtime suffices. For perfect hash table, we have $n >= k$. A hash function is *minimal* when its space cost is reduced to the minimum: $n = k$.

Usually time optimality is more important than space optimality, and fortunately it is easier to generate perfect hash functions than minimal perfect hash functions. Moreover, non-minimal perfect hash functions frequently execute faster than minimal ones in practice. This phenomenon occurs since searching a sparse keyword table increases the probability of locating a "null" entry, thereby reducing string comparisons. `gperf`'s default behavior generates near-minimal perfect hash functions for keyword sets. However, `gperf` provides many options that permit user control over the degree of minimality and perfection.

6.1.3 Simple Uses of Gperf

Gperf is a source generator, just as Flex, Bison, and others. It takes the list of your keywords as input, and produces a fast function recognizing them. As for Flex and Bison, the input syntax allows for a prologue, containing directives for `gperf` and possibly some user declarations and initializations, and an epilogue, typically additional functions:

```
%{
  user-prologue
%}
gperf-directives
%%
keywords
%%
user-epilogue
```

Example 6.2: *Structure of a Gperf Input File*

All the *keywords* are listed on separate lines. They do not need to be enclosed in double quotes, but if you intend to include special characters or commas, you may use the usual C string syntax. When run, `gperf` produces a C program on the standard output, including, in addition to your *user-prologue* and *user-epilogue*, two functions:

`static unsigned int` **hash** (`char *`*`string`*`, unsigned int `*`length`*`)` [Function]

Return an integer, named the *key*, characteristic of the *length* characters long C *string*.

`const char *` **in_word_set** (`const char *`*string*`, unsigned int` *length*) [Function]
If the C *string*, *length* character long, is one of the *keywords*, return a pointer to this keyword (i.e., not *string*, but the same content as *string*), otherwise return `NULL`.

For instance, the following simple Gperf input file is meant to recognize rude words, and to express its surprise on unknown words:

```
%{  /* -*- C -*- */
#include <stdio.h>
#include <stdlib.h>
%}
%%
sh*t
f*k
win*ows
Huh? What the f*?
%%
int
main (int argc, const char** argv)
{
  for (--argc, ++argv; argc; --argc, ++argv)
    if (in_word_set (*argv))
      printf ("I don't like you saying `%s'.\n", *argv);
    else
      printf ("Huh? What the f* `%s'?\n", *argv);
  return 0;
}
```

Example 6.3: *`rude-1.gperf' – Recognizing Rude Words With Gperf*

which we can try now:

```
$ gperf rude-1.gperf >rude.c
$ gcc -Wall -o rude rude.c
$ ./rude 'Huh? What the f*?'
Huh? What the f* `Huh? What the f*?'?
```

Huh? What the f* `Huh? What the f* `Huh? What the f*?'?'? It was supposed to recognize it!

You just fell into K&R, and it hurts. Our invocation of `in_word_set` above is wrong, we forgot to pass the length of the string, and since by default `gperf` produces K&R C, the compiler notices nothing (**FIXME:** *Pollux would like to see some actual output of Gperf here, what do you people think? It's roughly 100 lines, but I don't need them all..*). As a consequence, never forget to pass '`--language=ANSI-C`' to `gperf`. Just to check the result on our broken source:

```
$ gperf --language=ANSI-C rude.gperf >rude.c
$ gcc -Wall -o rude rude.c
error rude.c: In function `main':
error rude.c:91: too few arguments to function `in_word_set'
```

If we fix our invocation, '`in_word_set (*argv, strlen (*argv))`', then:

```
$ gperf --language=ANSI-C rude-2.gperf >rude.c
$ gcc -Wall -o rude rude.c
$ ./rude 'Huh? What the f*?'
I don't like you saying `Huh? What the f*?'.
```

To exercise it further:

```
$ ./rude 'sh*t' 'dear' '%%'
I don't like you saying 'sh*t'.
Huh? What the f* 'dear'?
I don't like you saying '%%'.
```

Huh? What the f* '%%'? You just fell into a bug in Gperf 2.7.2 which is a bit weak at parsing its input when the prologue includes solely user declarations, but no actual Gperf directive. You are unlikely to be bitten, but be aware of that problem.

But before going onto a more evolved example, let's browse some other features of Gperf.

6.1.4 Using Gperf

This section presents the most significant options of Gperf, the reader is invited to read the documentation of Gperf, section "Perfect Hash Function Generator" in *User's Guide to* `gperf` – *The GNU Perfect Hash Function Generator*, for all the details.

Instead of just returning a unique representative of each keyword, Gperf can be used to retrieve data associated to it efficiently.

'`--struct-type`'
'`-t`' Specify that the *gperf-prologue* include the definition of a structure, and that each line of the *keywords* section is an instance of this structure. The values of the members are separated by a comma. The function `in_word_set` will then return a pointer to the corresponding `struct`. This `struct` should use its first member to store the keyword, and name it `name`.

'`--initializer-suffix=`*initializers*'
'`-F` *initializers*'
When failing to produce a minimal table of keywords, and therefore falling back to a near-minimal table (see Section 6.1.2 [What Gperf is], page 130), `gperf` introduces empty entries of this `struct`, leaving the non *name* part unspecified. This can trigger spurious compiler warnings, in particular with `gcc`'s option '`-W`'. Quoting the GCC documentation:

> '`-W`' Print extra warning messages for these events: ... An aggregate has an initializer which does not initialize all members. For example, the following code would cause such a warning, because `x.h` would be implicitly initialized to zero:
>
> ```
> struct s { int f, g, h; };
> struct s x = { 3, 4 };
> ```

Use this option to specify the initializers for the empty structure members following the keyword. The *initializers* should start with a comma.

'`--omit-struct-type`'
'`-T`' Don't output the definition of the keywords' `struct`, it has been given only to describe it to Gperf. Use this option if your structure is defined elsewhere for the compiler.

By default `gperf` produces portable code, too portable actually: modern and useful features are avoided. The following options bring Gperf forward into the 21st century[2].

[2] Yet, in an effort to modernize, 8 bit characters are handled by default...

‘`--language=ANSI-C`’
‘`-l ANSI-C`’
: Output ANSI C instead of some old forgotten dialects of the previous century. You may also output C++, but decency prevents us from mentioning the other options.

‘`--readonly-tables`’
‘`-C`’
: Don’t be afraid to use `const` for internal tables. This is not only better style, it also helps some compilers to perform better optimization.

‘`--enum`’
‘`-E`’
: Output more C code than Cpp directives. In other words, use local `enum`s for Gperf internal constants instead of global `#define`.

‘`--switch=`*`total-switch-statements`*’
‘`-S` *`total-switch-statements`*’
: Instead of using internal tables, let the compiler perform the best job it can for your architecture by letting it face a gigantic `switch`. To assist compilers which are bad at long `switch`es, you may specify the depth of nested `switch` via *total-switch-statements*. Using 1 is fine with GCC.

You may want to include several Gperf outputs within a single application or even a single compilation unit. Therefore you need to avoid multiple uses of the same symbols. We already described ‘`--enum`’, and its usefulness to avoid global `#define`s.

‘`--hash-fn-name=`*`name`*’
‘`-H` *`name`*’ Specify the name of the `hash` function.

‘`--lookup-fn-name=`*`name`*’
‘`-H` *`name`*’ Specify the name of the `in_word_set` function.

Finally, many options allow to tune the hash function, see section “Options for changing the Algorithms employed by `gperf`” in *User’s Guide to* `gperf` *– The GNU Perfect Hash Function Generator*, for the exhaustive list. The most important options are:

‘`--compare-strncmp`’
‘`-c`’
: Use `strncmp` rather than `strcmp`. Using `strcmp` is the default because it is faster: it performs arithmetics operations on two items, the two strings, while `strncmp` additionally needs to maintain the length.

‘`--duplicates`’
‘`-D`’
: Handle collisions. When several keywords share the same hash value, they are said to *collide*. By default `gperf` fails if it didn’t find any collision free table. With this option, the *word* is compared to all the keywords that share its hash value, which results in degraded performances. Helping Gperf to avoid the collisions is a better solution if speed is your concern.

‘`--key-positions=`*`positions`*’
‘`-k` *`positions`*’
: By default Gperf peeks only at the first and last characters of the *keywords*; override this default with the comma separated list of *positions*. Each position may be a number, an interval, ‘`$`’ to designate the last character of each keyword, or ‘`*`’ to consider all the characters.

  This option helps solving collisions.

6.1.5 Advanced Use of Gperf

This section is devoted to a real application of Gperf: we will design `numeral`, an M4 module providing the builtin `numeral`, which converts numbers written in American English[3] into their actual value[4].

While it is obvious that we will need to map tokens (e.g., '`two`', '`forty`', '`thousand`'...) onto their values ('`2`', '`40`', '`1000`'...), the algorithm to reconstruct a value is less obvious. For a start, we won't try to handle "and" as in "one hundred and fifty".

Looking at '`two hundred two`' it is obvious that, reading from left to right, if the value of a token, '`two`', is smaller than that of the next token, '`hundred`', then we have to multiply, 2 * 100 = 200. If not, '`hundred`' followed by '`two`', then we need to add, 200 + 2. It seems that using two variables —`res` for the current result, and `prev_value` to remember the value of the previous token— is enough.

Unfortunately, it is not that simple. For instance '`two thousand two hundred`' would produce ((2 * 1000) + 2) * 100, while the correct equation is (2 * 1000) + (2 * 100). Since we are reading from left to right, we will need an additional variable to store the hundreds, `hundreds`. If the value of the current token is smaller that 1000, then we apply the previous algorithm to compute `hundreds` (add if smaller, multiply if greater), otherwise multiply the current `hundreds` by this value, and add to the global result.

```
if token value >= 1000 then
  res += hundreds * token value
  hundreds = 0
elseif previous value <= token value then
  hundreds *= token value
else
  hundreds += token value
end if
```

Finally, note that for '`hundreds *= token value`', `hundreds` should be initialized to 1, while in '`hundreds += token value`', 0 is the proper initialization. Instead of additional `if`s, we will use the GNU C extension, '*`foo ? : bar`*' standing for '*`foo ? foo : bar`*'[5].

Finally, let the file '`atoms.gperf`' be:

```
%{ /* -*- C -*- */

#if HAVE_CONFIG_H
#  include <config.h>
#endif

#include <m4module.h>
#include <stdint.h>
#include "numeral.h"

%}
```

[3] In American English, the words "billion", "trillion", etc. denote a different value than in Commonwealth English.

[4] To be more rigorous, *internally* they are converted into `uintmax_t`, and output as strings of digits.

[5] There are not strictly equivalent, since in '*`foo ? foo : bar`*' the expression *foo* is evaluated *twice*, while in '*`foo ? : bar`*' it is evaluated only once.

```
struct atom_s
{
  const char *name;
  const uintmax_t value;
};
%%
# Units.
zero,  0
one,   1
two,   2
three, 3
four,  4
five,  5
six,   6
seven, 7
eight, 8
nine,  9
# Teens.
ten,       10
eleven,    11
twelve,    12
thirteen,  13
fourteen,  14
fifteen,   15
sixteen,   16
seventeen, 17
eighteen,  18
nineteen,  19
# Tens.
twenty,   20
thirty,   30
forty,    40
fifty,    50
sixty,    60
seventy,  70
eighty,   80
ninety,   90
# Hundreds.
hundred,  100
hundreds, 100
```

```
# Powers of thousand.
thousands,    1000
thousand,     1000
million,      1000000
millions,     1000000
billion,      1000000000
billions,     1000000000
trillion,     1000000000000
trillions,    1000000000000
quadrillion,  1000000000000000
quadrillions, 1000000000000000
quintillion,  1000000000000000000
quintillions, 1000000000000000000
%%
uintmax_t
numeral_convert (const char *str)
{
  const char *alphabet = "abcdefghijklmnopqrstuvwxyz";
  uintmax_t res = 0, hundreds = 0, prev_value = 0;

  while ((str = strpbrk (str, alphabet)))
    {
      size_t len = strspn (str, alphabet);
      struct atom_s *atom = in_word_set (str, len);
      if (!atom)
        {
          numeral_error = strndup (str, len);
          return 0;
        }
      if (atom->value >= 1000)
        {
          res += (hundreds ? : 1) * atom->value;
          hundreds = 0;
        }
      else if (prev_value <= atom->value)
        {
          hundreds = (hundreds ? : 1) * atom->value;
        }
      else
        {
          hundreds += atom->value;
        }
      prev_value = atom->value;
      str += len;
    }
  return res + hundreds;
}
```

Example 6.4: '`atoms.gperf`' – *Converting Numbers in American English into Integers*

Finally, '`numeral.c`' contains all the needed hooks to create an M4 module, and the definition of the builtin, `numeral`:

```
/*--------------------.
| numeral(EXPRESSION) |
`--------------------*/

M4BUILTIN_HANDLER (numeral)
{
  uintmax_t res;
  char buf[128];

  res = numeral_convert (M4ARG (1));
  if (!numeral_error)
    {
      sprintf (buf, "%ju", numeral_convert (M4ARG (1)));
      obstack_grow (obs, buf, strlen (buf));
    }
  else
    {
      M4ERROR ((warning_status, 0,
                _("Warning: %s: invalid number component: %s"),
                M4ARG (0), numeral_error));
      free (numeral_error);
      numeral_error = NULL;
    }
}
```

Example 6.5: '`numeral.c`' – *M4 Module Wrapping* '`atoms.gperf`'

Let us run `gperf` to create '`atoms.c`':

```
$ gperf --switch=1 --language=ANSI-C --struct-type atoms.gperf >atoms.c
error Key link: "eight" = "three", with key set "et".
error Key link: "thirty" = "twenty", with key set "ty".
error Key link: "fifty" = "forty", with key set "fy".
error Key link: "trillion" = "thirteen", with key set "nt".
error Key link: "trillions" = "thousands", with key set "st".
error Key link: "quintillion" = "quadrillion", with key set "nq".
error Key link: "quintillions" = "quadrillions", with key set "qs".
error 7 input keys have identical hash values,
error try different key positions or use option -D.
```

Eek! Gperf recognizes that its simple heuristic, based apparently on peeking only at the first and last characters of each word (and their lengths), fails, and it lists the clashes. If performances do not matter too much for your application, such as our, then using '`--duplicates`', '`-D`', asks Gperf to handle the collisions using successive string comparisons:

```
$ gperf -S 1 -L ANSI-C -t -D atoms.gperf >atoms.c
error 7 input keys have identical hash values, examine output carefully...
```

This time the message is only informative, the hash function will just be efficient, not extremely efficient, that's all. If, on the contrary, performances matter to you, then you may play with the various options of `gperf`, see Section 6.1.4 [Using Gperf], page 133. We will try to find a better set of character positions to peek at, using '`--key-positions`', '`-k`'.

Notice that the last characters are often not very informative: '`n`' is frequently used because of '`-teen`' and '`-lion`', '`y`' because of '`yty`', and '`s`' because of the plurals etc. This suggests not

to use '$' in our key positions. Since 'ten' is our shortest word, we need to consider at least one of 1, 2, and 3 as key position. If you try each one, 3 is the best choice with 7 collisions, against 13 for 1, and 14 for 2:

```
$ gperf -t -k3 atoms.gperf >/dev/null
error Key link: "twelve" = "eleven", with key set "e".
error Key link: "twenty" = "eleven", with key set "e".
error Key link: "forty" = "three", with key set "r".
error Key link: "hundreds" = "nineteen", with key set "n".
error Key link: "billion" = "million", with key set "l".
error Key link: "billions" = "millions", with key set "l".
error Key link: "trillion" = "thirteen", with key set "i".
error 7 input keys have identical hash values,
error try different key positions or use option -D.
```

Choosing 1 is seducing, since most of the collided words start with a different character:

```
$ gperf -t -k1,3 atoms.gperf >/dev/null
error Key link: "twenty" = "twelve", with key set "et".
error Key link: "trillion" = "thirteen", with key set "it".
error 2 input keys have identical hash values,
error try different key positions or use option -D.
```

Finally, obviously, the fourth, fifth and sixth characters will solve the remaining conflicts. Choosing the fourth, we can run `gperf` with all the options required by our application:

```
$ gperf -S 1 -L ANSI-C -t -k1,3,4 atoms.gperf >atoms.c
```

then compile our numeral module, and try it:

```
$ m4 -m numeral
numeral(forty)
⇒40
numeral(two)
⇒2
numeral(forty-two)
error m4: stdin: 3: Warning: numeral: invalid number component: forty
⇒
```

Huh? What the f* '`invalid number component: forty`'? Clearly this Gperf recognizer demonstrated that it does know '`forty`' and '`two`', but it does not recognize `forty` in `forty-two`. How come?

We just fell into a huge trap left wide open in Gperf, meant by its authors: *although* we do specify the length of the word to check, *Gperf uses* `strcmp`! In the present case, it is comparing '`forty-two`' against '`forty`' using `strcmp`, even though we did mention the five first characters of '`forty-two`' were to be checked.

In fact, what we told Gperf is that if it wants to peek at the last character of the word, then this character is at the fifth position, but when comparing strings to make sure the word and the keyword it might be are equal, it uses a plain regular C string comparison.

I cannot emphasize this too much: *if you are not submitting 0 ended strings then use '*`--compare-strncmp`*'.*

Trying again:

```
$ gperf -S 1 -L ANSI-C -t -k1,3,4 --compare-strncmp atoms.gperf >atoms.c
```

and then:

```
$ m4 -m numeral
numeral(forty-two)
```

```
⇒42
numeral(
                  twelve quintillion
three hundred forty-five quadrillion
  six hundred seventy-eight trillion
            nine hundred one billion
     two hundred thirty-four million
   five hundred sixty-seven thousand
                eight hundred ninety
)
⇒12345678901234567890
```

6.1.6 Using Gperf with the GNU Build System

Currently neither Autoconf nor Automake provide direct Gperf support, but interfacing Gperf with them is straightforward.

All '`configure.ac`' needs to do is to look for `gperf` and to provide its definition for Makefiles via `AC_SUBST` (**FIXME:** *ref Autoconf.*). In fact, using Automake's macro to put `gperf` under the control of `missing` is enough:

```
AM_MISSING_PROG([GPERF], [gperf])
```

Then your '`Makefile.am`' should include:

```
# Handling the Gperf code
GPERFFLAGS = --compare-strncmp --switch=1 --language=ANSI-C
BUILT_SOURCES = atoms.c

atoms.c: atoms.gperf
        if $(GPERF) $(GPERFFLAGS) --key-positions=1,3,4 --struct-type \
           atoms.gperf >$@t; then \
          mv $@t $@; \
        elif $(GPERF) --version >/dev/null 2>&1; then \
          rm $@t; \
          exit 1; \
        else \
          rm $@t; \
          touch $@; \
        fi
```

I personally avoid using short options in scripts and Makefiles, because short options are likely to change and because long options are easier to understand when you don't know the program. Do not forget to help Automake understand that '`atoms.c`' is to be built early (before its uses) using `BUILT_SOURCES`.

The program `gperf` is a *maintainer requirement*: someone changing the package needs it, but its result is shipped so that a regular user does not need it. Hence, we go through some hoops in order to ensure that a failed run of `gperf` doesn't erase the maintainer's pre-built copy of '`atoms.c`'. Had `gperf` provided an '`--output`' option, `missing` would have handled these details gracefully.

There are three cases to handle:

`gperf` succeeded
: Then just rename the temporary output file, '`@t`', as the actual output, '`@`';

`gperf` failed
: If the `$(GPERF)` invocation failed, but '`$(GPERF) --version`' succeeded, then this is certainly an actual error in the input file. In this case, do not hide the failure and exit with failure.

`gperf` is missing
: If `$(GPERF)` does not answer to '`--version`', it is certainly missing, and `missing` already suggested to install Gperf. Then remove the temporary output file, and let the compilation proceed by updating the timestamp of the output file. That's a best effort, essentially helping users who get the project with broken timestamps.

6.1.7 Exercises on Gperf

In this section, we address some issues left opened by the previous section, Section 6.1.5 [Advanced Use of Gperf], page 135.

Testing
: Write an Autotest test suite for your `numeral` module. See Chapter 12 [Software Testing with Autotest], page 207, for all the details on designing and implementing test suites

Overflow
: Augment the previous algorithm with overflow detection.

Non Standard Numbers
: The algorithm presented above produces invalid results when the numbers are presented in a perfectly human understandable form, but nonstandard. For instance 2 000 2 000 000 000 is to be said "two trillion two billion" but people would understand "two trillion two thousand millions", which our module does not recognize properly:

```
$ m4 -m numeral
numeral(two trillion two thousand millions)
⇒2000001002000
```

This phenomenon is the same as we already observed with hundreds, see Section 6.1.5 [Advanced Use of Gperf], page 135. Hint: a stack might be helpful.

Invalid Numbers
: Try to diagnose invalid numbers, which humans would reject. For instance:

```
$ m4 -m numeral
numeral(forty-eleven)
⇒51
```

Hint: without an actual grammar (**FIXME:** *ref to Bison.*), it might not be possible, or at least, extremely clumsy.

6.2 Scanning with Flex

6.2.1 Looking for Tokens

When processing texts written in formal languages (such as programming languages, mark up languages, etc.), finding keywords is not enough: you also need to recognize numbers, identifiers, strings etc. We will name *token* or *lexeme* a series of characters which must be grouped together into a "word". For instance the following C snippet:

```
const char *cp = "Foo";
```

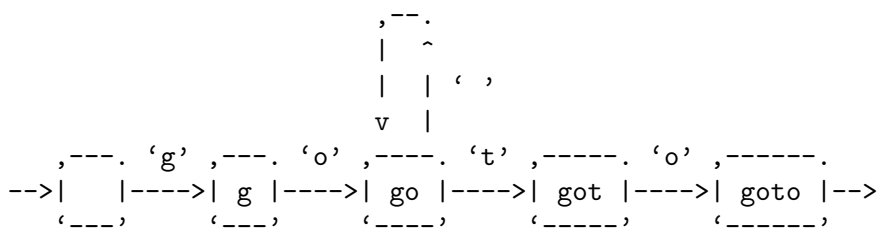
is composed of seven tokens: ‘`const`’, ‘`char`’, ‘`*`’, ‘`cp`’, ‘`=`’, ‘`"Foo"`’, and ‘`;`’. We excluded the white characters because they are not relevant in C, their purpose is limited to separating the tokens (compare ‘`const char`’ to ‘`constchar`’).

Keywords, operators (which are nothing else but non alphabetic keywords) are tokens, and we saw that Gperf is a fine tool to recognize keywords, see Section 6.1 [Scanning with Gperf], page 127. Nevertheless, as demonstrated in the `numeral` example, it does not help us segmenting the input into tokens, which we handled thanks to `strpbrk` and `strspn` (see *example 6.4*). A better tool could have assisted us in such a task.

Contrary to the keywords, there are infinitely many tokens: you may write infinitely many different strings, infinitely many identifiers etc. Again, we are hitting a limitation of Gperf: it will never be able to recognize strings.

Worse yet! It cannot help us recognize some keywords. Some languages allow for `goto` written with arbitrarily many spaces between `go` and `to`! Of course, we could teach it `goto` on the one hand, and then `go` and `to` on the other hand, but it would be so much easier to be able to specify that a token GOTO can be written as ‘`go`’, then any number of spaces including none, then ‘`to`’...

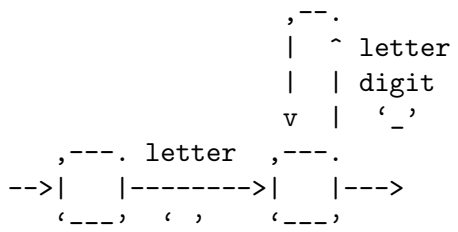
Clearly, the technology used by Gperf cannot answer such a problem: peeking at fixed places within a string no longer makes sense if the string can have any length. Nevertheless, we exposed a technique that might help, see *example 6.1*. If we relax the constraint of building a tree, i.e., if we allow cycles in our construct, then we can build a fast GOTO recognizer:



Example 6.6: *A Fast* GOTO *Recognizer*

Such small recognizers are named *Finite State Recognizers*[6], FSR, each node being named a *state*, and each labeled arrow a *transition*.

It is one of the most beautiful results in computer size that one can always write an FSR for all the keywords, identifiers, numbers etc. For instance, identifiers usually look like:

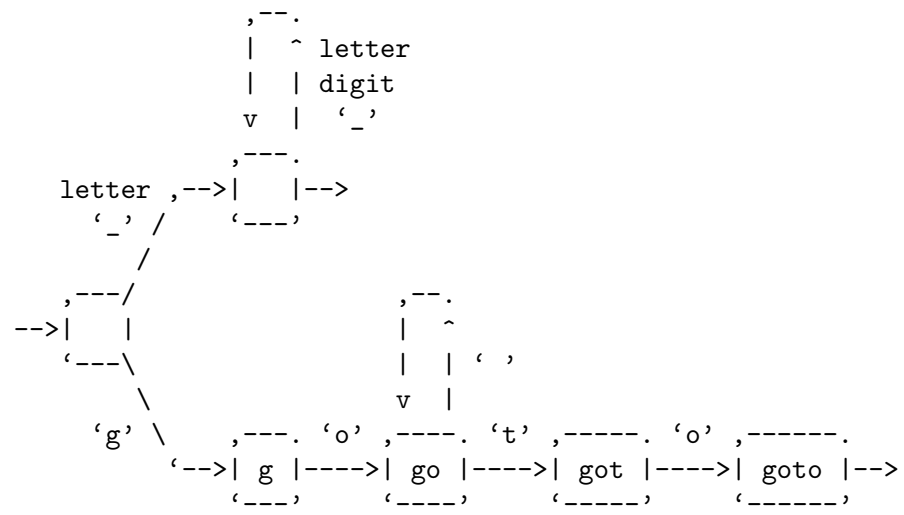


Example 6.7: *A Fast Identifier Recognizer*

In other words, it must start with one letter or an underscore, and can be followed by any number of letter, digit or underscore, including zero.

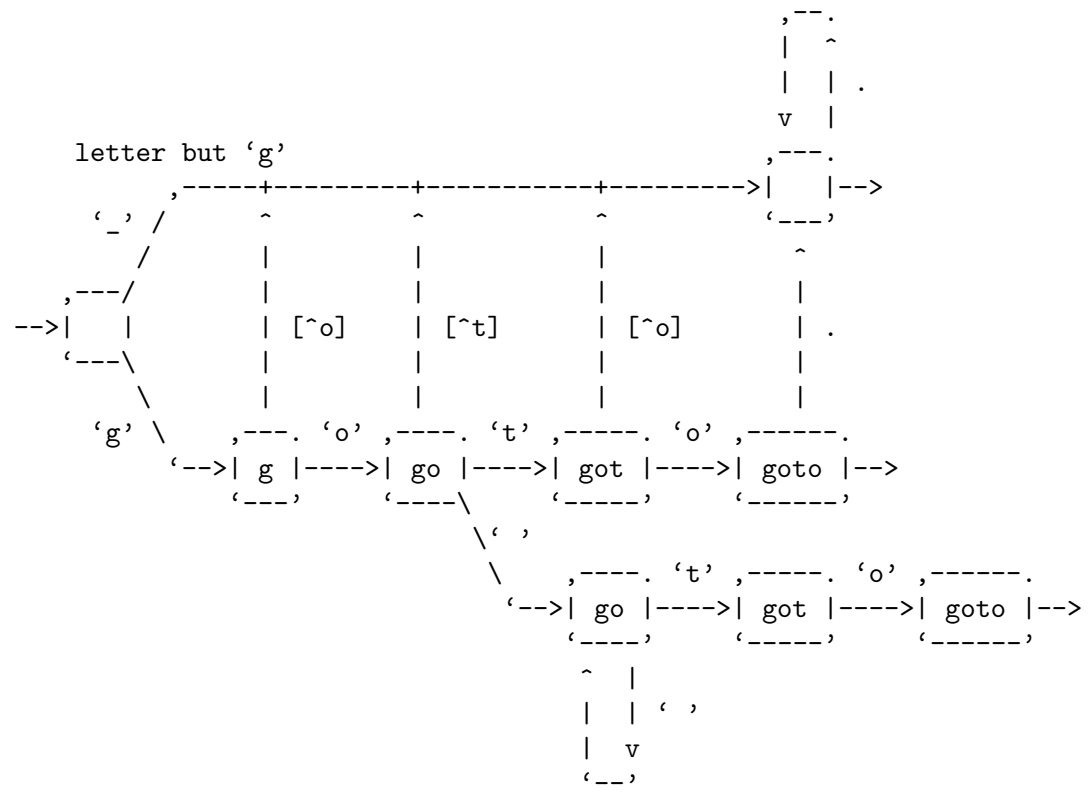
All this independent FSR can be grouped together into a bigger one:

[6] You may also find *Finite State Machine*, or *Automaton*, which is less specific since not all automata are recognizers: some are *generators* and generate words, and others are *transducers* and transform words.



Example 6.8: *A Fast Nondeterministic* GOTO *and Identifier Recognizer*

Obviously there is a competition between the two branches of this FSR: from the initial state, upon '`g`', it may go in either the identifier or GOTO branch. This FSR is said to be *nondeterministic*: it may perform choices at runtime. It recognizes a word if and only if one of its possible executions reaches an exiting arrow. It is another most beautiful result in computer science that there is a deterministic FSR that recognize exactly the same set of words. Better yet: one can build it from the nondeterministic automaton:



Example 6.9: *A Fast Deterministic* GOTO *and Identifier Recognizer*

where '`.`' stands for "any letter, digit or underscore", and '`[^o]`' for "any letter, digit or underscore but '`o`'". This deterministic automaton is a very efficient token recognizer.

Our goal is ultimately to design a tool that can build this automaton for us, therefore we need a convenient notation to describe the tokens. It turns out that regular expressions are as powerful as FSRs are. In other words, the language, the set of words recognized by an automaton can always be described by a regular expression, and conversely, for any regular expression there exists a deterministic FSR which recognizes its language.

For instance, '`goto`' is a perfect description of the word '`goto`': in regular expressions most characters represent themselves. In addition, the star, '`*`', is used to denote the repetition of the previous regular expression. For instance, '` *`' denotes any number of spaces, including none; the regular expression '`go *to`' denotes the same language as that recognized by the automaton of *example 6.6*.

We also need to express a choice, for instance, to be able to describe that all the identifiers start with a letter or an underscore. The operator '`|`' denotes the alternative, and as usual, parentheses, '`( )`', allow for changing the precedence. For instance '`a|b|c|d|e|f`' denotes one character amongst '`a`', '`b`', '`c`', '`d`', '`e`', '`f`', '`a|b|c|d|e|f*`' represent a single '`a`', or a '`b`', or a '`c`', or a '`d`', or a '`e`', or any number of '`f`' including none, and '`(a|b|c|d|e|f)*`' denotes any string written with '`a`', '`b`', '`c`', '`d`', '`e`', '`f`', including the empty string.

This notation is inconvenient for character alternatives, therefore we introduce some abbreviations: '`[abcdef]`' stands for '`(a|b|c|d|e|f)`', and '`[a-f]`' stands for '`[abcdef]`'. Therefore, the language recognized by the automaton of the *example 6.7* can be represented as '`[a-zA-Z_][a-zA-Z0-9_]*`'. Finally, '`go *to|[a-zA-Z_][a-zA-Z0-9_]*`' denotes the language recognized by the FSRs of *example 6.8* and of *example 6.9*.

When recognizing a keyword, getting a structure associated to it, as with Gperf, see Section 6.1.5 [Advanced Use of Gperf], page 135, is very convenient, but in the case an identifier, you want to know (i) it is an identifier, and (ii), which one. When reading an integer, i.e., a word written with digits, you don't want to get its textual representation, but rather a genuine `int` containing its value. Finally, you don't even want to hear about spaces, since they are just separators.

Therefore, it will be more convenient to associate *actions* to regular expressions, rather than structures. These actions will rely on `return` to declare that something was recognized and to say what; typically there will be no `return` associated to spaces since they are meaningless. To cope with identifiers and integers which, in addition to their type, have a value, these actions will be provided with two variables, `yytext` pointing to the beginning of the string which was recognized, and `yyleng`, holding its length. Then the action is free to compute whatever is wanted, for instance converting a string of digits into an actual `int`, and then to provide the environment with it via, for instance, a global variable. It is customary to name this global `yylval`.

Each pair of regular expression/action will be listed one a single line, separated by spaces. Using braces will allow for longer actions, and using '`" "`' allow for denoting a space which is not a separator.

It will be more convenient for us if it read directly from a file, the standard input by default, instead of a string. It then needs a means to inform us that it finished reading the input; returning 0 will do, but keep this in mind when numbering the tokens. Finally, recognition and actions will be embedded in a routine, which we will name `yylex`. We will use the same file layout as in Gperf, see *example 6.2*.

```
%{
#include <stdio.h>
#include <stdlib.h>
const char *yylval = NULL;
enum token_e { token_goto = 1, token_identifier };
%}
%%
go" "*to                return token_goto;
[a-zA-Z_][a-zA-Z0-9_]*  yylval = yytext; return token_identifier;
" "                     /* Do nothing. */;
%%
int
main (void)
{
  enum token_e token;
  while ((token = yylex ()))
    if (token == token_goto)
      printf ("Saw a goto.\n");
    else if (token == token_identifier)
      printf ("Saw an identifier (%s).\n", yylval);
  printf ("End of file.\n");
  return 0;
}
```

Example 6.10: '`goid.l`' – *A* GOTO *and Identifier Flex Recognizer*

This turns out to be perfectly valid Flex source which we can immediately compile:

```
$ flex -ogoid.c goid.l
$ gcc -Wall -W goid.c -lfl -o goid
error goid.c:976: warning: 'yyunput' defined but not used
```

for the time being, forget about '`-lfl`' and the warning, just try `goid`:

```
$ echo 'gotoo goto go to' | ./goid
Saw an identifier (gotoo).
Saw a goto.
Saw a goto.

End of file.
```

You may wonder why '`gotoo`' is not recognized as a GOTO followed by the identifier '`o`'. Our input file relies on a fundamental rule in Lex matching: if several patterns match the current input, the longest wins, and if several patterns match the same number of characters, then the topmost one wins. This explains why the second '`goto`' is recognized as a GOTO and not as an identifier although it does look like one: always write the most generic rule last.

You may also wonder where the spurious empty line comes from. Flex provides a default action for any character which is not caught by the scanner: echoing it to its output. Our scanner has no rule for the newline character, hence it is output.

6.2.2 What Flex is

Lex is a generator of fast scanners. A *scanner* is a program or routine which recognizes tokens (identifiers, keywords, strings etc.) in texts (files or strings). Lex's input is composed of *rules*,

pairs of regular expressions and C code. The regular expression describes the patterns that the different tokens should follow to trigger the associated C code.

Lex is based on Finite States Recognizers, which are known to be extremely efficient.

Lex, and all its declinations (CAMLLex for CAML, Alex for Ada, JLex for Java etc.) are used in numerous applications: compilers, interpreters, batch text processing etc.

Flex is a free software implementation of Lex, as described by the POSIX standard. It is known to produce extremely efficient automata; many options are available to tune various parameters. It provides a wide set of additional options and features, produces portable code, supports a more pleasant syntax, and stands as a standard of its own. Since many Lex do have problems (inter-Lex compatibility and actual bugs), there is no reason to limit oneself to the Portable Lex subset of Flex: the portable C code produced by Flex can be shipped in the package and will compile cleanly on the user's machine. It imposes no restriction on the license of the produced recognizer.

6.2.3 Simple Uses of Flex

Flex is a source generator, just as Gperf, Bison, and others. It takes a list of regular expressions and actions, and produces a fast function triggering the action associated to the pattern recognized. As for Gperf and Bison, the input syntax allows for a prologue, containing Flex directives and possibly some user declarations and initializations, and an epilogue, typically additional functions:

```
%{
  user-file-prologue
%}
flex-directives
%%
%{
  user-yylex-prologue
%}
regular-expression-1    action-1
regular-expression-2    action-2
...
%%
user-epilogue
```

Example 6.11: *Structure of a Flex Input File*

All the pairs of *regular-expression* and *action* is listed on separate lines. The *regular-expression* must be written at the first column, otherwise it is considered as code to output inside the function which will be produced. This can be used to leave comments in the input. The *action*s maybe enclosed in braces if they are several lines long, and *action* = '`|`' stands for "same as the next action".

When run, `flex` produces a file named '`lex.yy.c`', containing a C program including, in addition to your *user-prologue* and *user-epilogue*, one function:

`int` **yylex** () [Function]

Scan the `FILE *yyin`, which defaults to the standard input, for tokens. Trigger the *action* associated to the succeeding *regular-expression*. Return 0 when it should no longer be called, typically when the end of the file is reached. Otherwise, typically returns the kind of the token that has just been recognized.

For instance, this simple Flex input file is meant to recognize rude words, and to express its surprise on unknown words:

```
%{  /* -*- C -*- */
%}
%%
"sh*t"                |
"f*k"                 |
"win*ows"             |
"Huh? What the f*?"   printf ("I don't like you saying `%s'.\n", yytext);
^.*$                  printf ("Huh? What the f* `%s'?\n", yytext);
\n                    /* Ignore. */
%%
```

Example 6.12: `rude-3.l` – *Recognizing Rude Words With Flex*

which we can try now:

```
$ flex -orude-3.c rude-3.l
$ gcc -Wall -o rude-3 rude-3.c -lfl
error rude-3.c:1020: warning: `yyunput' defined but not used
$ echo 'dear' | ./rude-3
Huh? What the f* `dear'?
```

We paid attention to writing the most general rules last, and to providing a rule to prevent the newline characters from being echoed to the standard output. This is needed because '`.`' does not match the newline characters, hence '`^.*$`' doesn't cover them.

You certainly have noted that we did not provide any `main`, however the program works: the Flex library, `libfl`, provides a default `main` which calls `yylex` until the end of file is found.

Let's try an actual match:

```
$ echo 'Huh? What the f*?' | ./rude-3
Huh? What the f* `Huh? What the f*?'?
```

Huh? What the f* '`Huh? What the f* `Huh? What the f*?'`'? It was supposed to recognize it!

You just fell onto so-called right contexts: '`$`' is exactly equivalent to '`/\n`' standing for "if followed by a newline". The bad news, is that when '`/`*`right-context`*' is used, the number of characters matched by *right-context* counts to elect the longest match (but of course doesn't get into `yyleng`). In other words, '`^.*$`' matched the whole line *plus* the newline, hence it wins over the dedicated pattern.

As much as possible you should avoid depending upon contexts, i.e., if you ever design a language, make sure it can be scanned without. If we simply replace '`^.*$`' with '`.*`' in the *example 6.12*, then we have:

```
$ echo 'Huh? What the f*?' | ./rude-4
I don't like you saying `Huh? What the f*?'.
```

6.2.4 Using Flex

See the *example 6.11* for the general layout of a Flex input file.

6.2.4.1 Flex Directives

Flex supports several directives, only a few of them being presented below, see (**FIXME:** *cite Flex documentation.*), for more information. Most of them have command line option equivalent, but in typical uses it is better to embed them within the file.

‘`%option debug`’
: Produce a scanner which can be traced. This introduce a variable, `yy_flex_debug`, which, when set to a non zero value, triggers tracing messages on the standard error output.

  You are encouraged to use this option, in particular when developing your scanner, and to have some option to set `yy_flex_debug`. In particular, never write `printf`-like tracing code in your scanner: that's an absolute waste of time.

‘`%option nodefault`’
: Die with an error message on unmatched characters instead of echoing them. We advise you not to rely on the default rule for sake of completeness, therefore, you should always use it to find holes in your rules.

‘`%option nounput`’
‘`%option noyywrap`’
: Specify your scanner does not use `unput` and/or `yywrap`. These two functions are beyond the scope of this book and won't be detailed. Nevertheless we present these options so that (i) we no longer need the Flex library (which provides a default `yywrap`), and (ii) our scanners compile without triggering ‘`warning: ‘yyunput’ defined but not used`’.

‘`%option outfile="`*file*`"`’
: Save the scanner in *file*.

‘`%option prefix="`*prefix*`"`’
: Replaces the default ‘`yy`’ prefix with *prefix*. It also changes the default output file from ‘`lex.yy.c`’ to ‘`lex.`*prefix*`.c`’.

6.2.4.2 Flex Regular Expressions

The characters and literals may be described by:

‘*x*’ the character *x*.

‘`.`’ any character except newline.

‘`[xyz]`’ Any characters amongst ‘`x`’, ‘`y`’ or ‘`z`’. You may use a dash for character intervals: ‘`[a-z]`’ denotes any letter from ‘`a`’ through ‘`z`’. You may use a leading hat to negate the class: ‘`[0-9]`’ stands for any character which is not a decimal digit, including new-line.

‘`\`*x*’ if *x* is an ‘`a`’, ‘`b`’, ‘`f`’, ‘`n`’, ‘`r`’, ‘`t`’, or ‘`v`’, then the ANSI-C interpretation of ‘`\`*x*’. Otherwise, a literal ‘*x*’ (used to escape operators such as ‘`*`’).

‘`\0`’ a NUL character.

‘`\`*num*’ the character with octal value *num*.

‘`\x`*num*’ the character with hexadecimal value *num*.

‘`"`*string*`"`’
: Match the literal *string*. For instance ‘`"/*"`’ denotes the character ‘`/`’ and then the character ‘`*`’, as opposed to ‘`/*`’ denoting any number of slashes.

‘`<<EOF>>`’ Match the end-of-file.

The basic operators to make more complex regular expressions are, with *r* and *s* being two regular expressions:

‘`(r)`’ Match an *r*; parentheses are used to override precedence.

‘`rs`’ Match the regular expression *r* followed by the regular expression *s*. This is called *concatenation*.

‘`r|s`’ Match either an *r* or an *s*. This is called *alternation*.

‘`{abbreviation}`’
Match the expansion of the *abbreviation* definition. Instead of writing

```
%%
[a-zA-Z_][a-zA-Z0-9_]*   return IDENTIFIER;
%%
```

you may write

```
id  [a-zA-Z_][a-zA-Z0-9_]*
%%
{id}   return IDENTIFIER;
%%
```

The *quantifiers* allow to specify the number of times a pattern must be repeated:

‘`r*`’ zero or more *r*’s.

‘`r+`’ one or more *r*’s.

‘`r?`’ zero or one *r*’s.

‘`r{[num]}`’ *num* times *r*

‘`r{min,[max]}`’
anywhere from *min* to *max* (defaulting to no bound) *r*’s.

For instance ‘`-?([0-9]+|[0-9]*\.[0-9]+([eE][-+]?[0-9]+)?)`’ matches C integer and floating point numbers.

One may also depend upon the context:

‘`r/s`’ Match an *r* but only if it is followed by an *s*. This type of pattern is called *trailing context*. The text matched by *s* is included when determining whether this rule is the “longest match”, but is then returned to the input before the action is executed. So the action only sees the text matched by *r*. Using trailing contexts can have a negative impact on the scanner, in particular the input buffer can no longer grow upon demand. In addition, it can produce correct but surprising errors. Fortunately it is seldom needed, and only to process pathologic languages such as Fortran. For instance to recognize its loop keyword, ‘`do`’, one needs:

```
DO/[A-Z0-9]*=[A-Z0-9]*,
```

to distinguish ‘`DO1I=1,5`’, a for loop where ‘`I`’ runs from 1 to 5, from ‘`DO1I=1.5`’, a definition/assignment of the floating variable ‘`DO1I`’ to 1.5. See Section 12.1.2 [Fortran and Satellites], page 209, for more on Fortran loops and traps.

‘`^r`’ Match an *r* at the beginning of a line.

‘`r$`’ Match an *r* at the end of a line. This is rigorously equivalent to ‘`r/\n`’, and therefore suffers the same problems, see Section 6.2.3 [Simple Uses of Flex], page 146.

6.2.4.3 Flex Actions

When a pattern is matched, the corresponding action is triggered. The actions default to nothing, i.e., discard the current token. In the output, the actions are embedded in the `yylex` function, therefore, using `return` sets the return value of `yylex`.

All the actions may use:

`const char * yytext` [Variable]
`size_t yyleng` [Variable]
Pointer to the beginning and length of the input string that matched the regular expression.

`|` [Macro]
Stands for "same as the next action".

ECHO [Macro]
Propagate the token to `FILE *yyout`, which defaults to the standard output. This is the default action.

YY_USER_INIT [Macro]
If defined, evaluated at the first invocation of `yylex`. This macro can be used to perform some initialization of your scanners: it will be execute as soon as you start them.

But be aware that if you run your scanner on several inputs, then only the first run will trigger this code. If you need to perform some initialization at each new file, you will have to find some other means.

YY_USER_ACTION [Macro]
Executed each time a rule matches, after `yytext` and `yyleng` were set, but before the action is triggered.

user-yylex-prologue [Macro]
Executed each time `yylex` is invoked. See *example 6.11* for its definition.

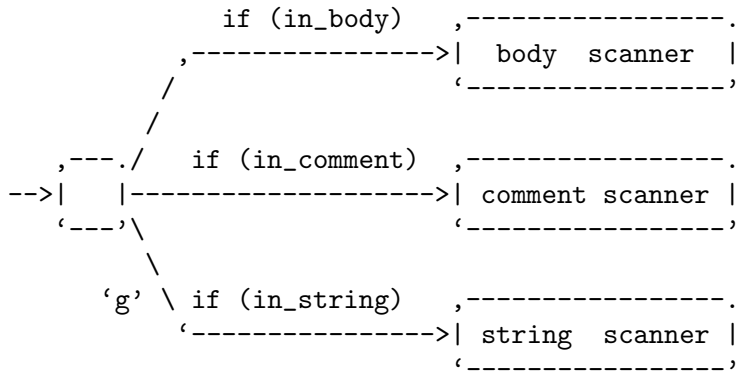
6.2.5 Start Conditions

Non keywords often need some form of conversion: strings of digits are converted into integers, and so on. This conversion often involves another scanning of the token, for instance to convert the escapes, e.g., '`\n`', into character literals. Writing this scanner by hand is easy, but frustrating.

Sometimes one is limited by the theory itself: imagine your language supports nested comments. It is easily proven that a language of balanced parentheses[7] cannot be described by regular expressions. Indeed, this would imply the existence of an FSR, with say q states. Then if we overflow its memory with more than q opening parentheses, it completely loses its count. Therefore there cannot be such an FSR, hence no regular expression, thus we are stuck! Nevertheless it would have been very easy to write a scanner solely tracking '`/*`' and '`*/`' and throwing away any other string.

Our scanners are nothing but automata, such as in the *example 6.9*. We could solve the two problems above simply if we could join the corresponding FSR to a new initial state labelled *with some conditions*:

[7] "Balanced parentheses" is to be understood in its broadest sense: including `begin/end`, '`/*`'/'`*/`' etc.



Example 6.13: *A Condition Driven* FSR *Combination*

These are called *start conditions*. They allow to combine small parsers into a bigger one. The default start condition is named `INITIAL`, others can be introduced thanks to the Flex directive `%x` *`start-condition`*.... To set the current start condition, i.e., to select the eligible branch *at the next run of the automaton*, use '`BEGIN` *`start-condition`*'. This is not a form of `return` or `goto`, the execution proceeds normally in the current action.

Finally, to complete the description of the rules by their conditions, use either

```
<start-condition, ...>pattern action
```

or

```
<start-condition, ...>{
  pattern-1 action-1
  pattern-2 action-2
}
```

6.2.6 Advanced Use of Flex

In this section we will develop a scanner for arithmetics, which will later be used together with a Bison generated parser to implement an alternative implementation of M4's `eval` builtin, see (**FIXME:** *Ref Bison, ylparse.y.*). Our project is composed of:

'`yleval.h`' a header common to all the files,

'`ylscan.l`' the scanner for arithmetics

'`ylparse.y`' the parser for arithmetics (**FIXME:** *ref.*).

'`yleval.c`' the driver for the whole module (**FIXME:** *ref.*).

Because locations are extremely important in error messages, we will look for absolute preciseness: we will not only track the line and column where a token starts, but also where it ends. Maintaining them by hand is tedious and error prone, so we will insert actions at appropriate places for Flex to maintain them for us. We will rely on Bison's notion of location:

```
typedef struct yyltype
{
  int first_line, first_column, last_line, last_column;
} yyltype;
```

which we will handle thanks to the following macros:

LOCATION_RESET (*location*) [Macro]

Initialize the *location*: first and last cursor are set to the first line, first column.

LOCATION_LINE (*location, num*) [Macro]
Advance the end cursor of *num* lines, and of course reset its column. A macro `LOCATION_COLUMN` is less needed, since it would consist simply in increasing the `last_column` member.

LOCATION_STEP (*location*) [Macro]
Move the start cursor to the end cursor. This is used when we read a new token. For instance, denoting the start cursor `S` and the end cursor `E`, we move from

```
1000 + 1000
^  ^
S  E
```

to

```
1000 + 1000
   ^
  S=E
```

LOCATION_PRINT (*file, location*) [Macro]
Output a human readable representation of the *location* to the stream *file*. This hairy macro aims at providing simple locations by factoring common parts: if the start and end cursors are on two different lines, it produces '`1.1-2.3`'; otherwise if the location is wider than a single character it produces '`1.1-3`', and finally, if the location designates a single character, it results in '`1.1`'.

Their code is part of '`yleval.h`':

```
/* Initialize LOC. */
# define LOCATION_RESET(Loc)                    \
  (Loc).first_column = (Loc).first_line = 1;  \
  (Loc).last_column =  (Loc).last_line = 1;

/* Advance of NUM lines. */
# define LOCATION_LINES(Loc, Num)               \
  (Loc).last_column = 1;                        \
  (Loc).last_line += Num;

/* Restart: move the first cursor to the last position. */
# define LOCATION_STEP(Loc)                     \
  (Loc).first_column = (Loc).last_column;     \
  (Loc).first_line = (Loc).last_line;

/* Output LOC on the stream OUT. */
# define LOCATION_PRINT(Out, Loc)                                  \
  if ((Loc).first_line != (Loc).last_line)                         \
    fprintf (Out, "%d.%d-%d.%d",                                   \
             (Loc).first_line, (Loc).first_column,                 \
             (Loc).last_line, (Loc).last_column - 1);              \
  else if ((Loc).first_column < (Loc).last_column - 1)             \
    fprintf (Out, "%d.%d-%d", (Loc).first_line,                    \
             (Loc).first_column, (Loc).last_column - 1);           \
  else                                                             \
    fprintf (Out, "%d.%d", (Loc).first_line, (Loc).first_column)
```

Example 6.14: '`yleval.h`' *(i) – Handling Locations*

Because we want to remain in the 'yleval_' name space, we will use %option prefix, but this will also rename the output file. Because we use Automake which expects flex to behave like Lex, we use %option outfile to restore the Lex behavior.

```
%option debug nodefault noyywrap nounput
%option prefix="yleval_" outfile="lex.yy.c"

%{
#if HAVE_CONFIG_H
#  include <config.h>
#endif
#include <m4module.h>
#include "yleval.h"
#include "ylparse.h"
```

Example 6.15: 'ylscan.l' – *Scanning Arithmetics*

Our strategy to track locations is simple, see Section 6.2.4.3 [Flex Actions], page 150. Each time yylex is invoked, we move the first cursor to the last position thanks to the *user-yylex-prologue*. Each time a rule is matched, we advance the ending cursor of yyleng characters, except for the rule matching a new line. This is performed thanks to YY_USER_ACTION. Each time we read insignificant characters, such as white spaces, we also move the first cursor to the latest position. This is done in the regular actions:

```
/* Each time we match a string, move the end cursor to its end. */
#define YY_USER_ACTION  yylloc->last_column += yyleng;
%}
%%
%{
  /* At each yylex invocation, mark the current position as the
     start of the next token.  */
  LOCATION_STEP (*yylloc);
%}
  /*  Skip the blanks, i.e., let the first cursor pass over them.  */
[\t ]+     LOCATION_STEP (*yylloc);
\n+        LOCATION_LINES (*yylloc, yyleng); LOCATION_STEP (*yylloc);
```

The case of the keywords is straightforward and boring:

```
"+"        return PLUS;
"-"        return MINUS;
"*"        return TIMES;
...
```

Integers are more interesting: we use strtol to convert a string of digits into an integer. The result is stored into the member number of the variable yylval, provided by Bison via 'ylparse.h'. We support four syntaxes: '10' is decimal (equal to... 10), '0b10' is binary (2), '010' is octal (8), and '0x10' is hexadecimal (16). Notice the risk of reading '010' as a decimal number with the naive pattern '[0-9]+'; you can either improve the regular expression, or rely on the order of the rules[8]. We chose the latter.

[8] Note that the two solutions proposed are *not* equivalent! Spot the difference between

```
[1-9][0-9]*  return NUMBER;
0[0-7]+      return NUMBER;
```

and

```
  /* Binary numbers. */
0b[01]+   yylval->number = strtol (yytext + 2, NULL, 2); return NUMBER;
  /* Octal numbers. */
0[0-7]+   yylval->number = strtol (yytext + 1, NULL, 8); return NUMBER;
  /* Decimal numbers. */
[0-9]+    yylval->number = strtol (yytext, NULL, 10); return NUMBER;
  /* Hexadecimal numbers. */
0x[:xdigit:]+ yylval->number = strtol (yytext + 2, NULL, 16); return NUMBER;
```

Finally, we include a catch-all rule for invalid characters: report an error but do not return any token. In other words, invalid characters are neutralized by the scanner:

```
  /* Catch all the alien characters. */
.   {
      yleval_error (yylloc, yycontrol, "invalid character: %c", *yytext);
      LOCATION_STEP (*yylloc);
    }
%%
```

where `yleval_error` is a variadic function (as is `fprintf`) and `yycontrol` a variable that will be both defined later.

This scanner is complete, it merely lacks its partner: the parser. But this is yet another chapter...

6.2.7 Using Flex with the GNU Build System

Autoconf and Automake provide some generic Lex support, but no Flex dedicated support. In particular Automake expects '`lex.yy.c`' as output file, which makes it go through various hoops to prevent several concurrent invocations of `lex` to overwrite each others' output.

As Gperf, Lex and Flex are maintainer requirements: someone changing the package needs them, but their result is shipped so that a regular user does not need them. I have already emphasized that you should not bother with vendor Lexes, Flex alone will fulfill all your needs, and keep you away from troubles. If you use Automake's `AM_PROG_LEX`, then if `flex` or else `lex` is found, it is used as is, otherwise `missing` is used to invoke `flex`. Unfortunately, if `lex` is available, but not good enough to handle your files, then the output scanner will be destroyed. Therefore, to avoid this, we really want to wrap the `lex` invocations with `missing`. I suggest:

```
# Sometimes Flex is installed as Lex, e.g., NetBSD.
AC_CHECK_PROG([FLEX], [flex lex], [flex])
# Force the use of 'missing' to wrap Flex invocations.
AM_MISSING_PROG([LEX], [$FLEX])
# Perform all the tests Automake and Autoconf need.
AM_PROG_LEX
```

Then, in your '`Makefile.am`', forget your Flex sources need a special handling, Automake takes care of it all. Merely list them as ordinary source files:

```
LDFLAGS = -no-undefined

pkglibexec_LTLIBRARIES = yleval.la
yleval_la_SOURCES = ylparse.y ylscan.l yleval.c yleval.h ylparse.h
```

```
[0-9]+    return NUMBER;
0[0-7]+   return NUMBER;
```

```
yleval_la_LDFLAGS = -module
```

and that's all. In particular, do not bother with `LEXLIB` at all: `configure` defines it to '`-ll`' or '`-lfl`', but we already emphasized that Flex' output is self-contained and portable, see Section 6.2.4 [Using Flex], page 147.

6.2.8 Exercises on Flex

Free Radix

The scanner we described knows four different input radices for numbers: decimal, binary, octal, and hexadecimal. M4 supports a fifth mode for arbitrary *radix* between 2 and 36: '`0r`*radix*`:`*number*'. Implement this mode in '`ylscan.l`'.

C Source Statistics

Using Flex, implement a clone of `csize`, a simple program performing statistics on C sources. For instance running it on the Yleval module gives:

```
$ csize -h yleval.h yleval.c ylscan.c ylparse.h ylparse.c
 total    blank lines w/   nb, nc    semi- preproc. file
 lines    lines comments    lines   colons  direct.
------+--------+--------+--------+--------+--------+----
    69       10       27       33        8        9 yleval.h
   237       35       37      165       60       13 yleval.c
  1730      296      279     1185      391      303 ylscan.c
    54        7        1       47        7       36 ylparse.h
  1290      178      237      915      288      279 ylparse.c
  3380      526      581     2345      754      640 total
```

C Strings Extend the previous program with statistics on C strings.

Beswitch We have already studied keyword recognizers, see Section 6.1.1 [Looking for Keywords], page 127. In particular, while Gperf is based on hash tables, we showed how a similar application could be based on a cascade of `switch`. Implement one such program, `beswitch`, thanks to Flex. `beswitch` must be a drop-in replacement of Gperf.

Your scanner must be better than that of Gperf, in particular, it shall not be confused by '`/* %} */`' or '`" %} "`' in the prologue. You are likely to need several start conditions, such as `C_CODE`, `PROLOGUE`, `COMMENT`, `STRING`, etc. The simple `BEGIN` will no longer be sufficient, and you will probably need some form of sub-scanner recursive calls: see `%option stack`, `yy_stack_push`, and `yy_stack_pop` in (**FIXME:** *Flex ref.*).

7 Parsing

7.1 Looking for Balanced Expressions

We have seen that Flex supports abbreviations, see Section 6.2.4.2 [Flex Regular Expressions], page 148. Using a more pleasant syntax, we could write for instance:

```
digit:  [0-9];
number: digit+;
```

It is then tempting to describe possibly parenthesized expressions using these abbreviations:

```
expr: '(' expr ')' | number
```

Example 7.1: *A Simple Balanced Expression Grammar*

to describe numbers nested in balanced parentheses.

Of course, Flex abbreviations cannot be recursive, since recursion can be used as above to describe balanced parentheses, which fall out of Flex' expressive power: Flex produces finite state recognizers, and FSR cannot recognized balanced parentheses because they have finite memory (see Section 6.2.5 [Start Conditions], page 150).

This suggests that we need some form of virtually infinite memory to recognize such a language. The most primitive form of an infinite memory device is probably stacks, so let's try to design a recognizer for `expr` using a stack. Such automata are named *pushdown automata*.

Intuitively, if the language was reduced to balanced parentheses without any nested number, we could simply use the stack to push the opening parentheses, and pop them when finding closing parentheses. Slightly generalizing this approach to the present case, it seems a good idea to push the tokens onto the stack, and to pop them when we find what can be done out of them:

Step	Stack	Input
1.		`( ( number ) )`
2.	`(`	`( number ) )`
3.	`( (`	`number ) )`
4.	`( ( number`	`) )`

At this stage, our automaton should recognize that the `number` on top of its stack is a form of `expr`. It should therefore replace `number` with `expr`:

5.	`( ( expr`	`) )`
6.	`( ( expr )`	`)`

Now, the top of the stack, '`( expr )`', clearly denotes an `expr`, according to our rules:

7.	`( expr`	`)`
8.	`( expr )`	
9.	`expr`	

Finally, we managed to recognize that '`( ( number ) )`' is indeed an expression according to our definition: the whole input has been processed (i.e., there remains nothing in the input), and the stack contains a single nonterminal: `expr`. To emphasize that the whole input must be processed, henceforth we will add an additional token, '`$`', standing for end of file, and an additional rule:

```
$axiom: expr $;
```

If you look at the way our model works, there are basically two distinct operations to perform:

shift Shifting a token, i.e., fetching the next token from the input, and pushing it onto the stack. Steps performed from the states 1, 2, 3, 5, and 7 are shifts.

reduce Reducing a rule, i.e., recognizing that the top of the stack represents some right hand size of a rule, and replacing it with its left hand side. For instance at stage 4, the top stack contains '`number`' which can be reduced to '`expr`' thanks to the rule '`expr: number;`'. At stages 6 and 8, the top of the stack contains '`( expr )`', which, according to '`expr: '(' expr ')';`' can be reduced to '`expr`'.

The initial rule, '`$axiom: expr $;`' is special: reducing it means we recognized the whole input. This is called *accepting*.

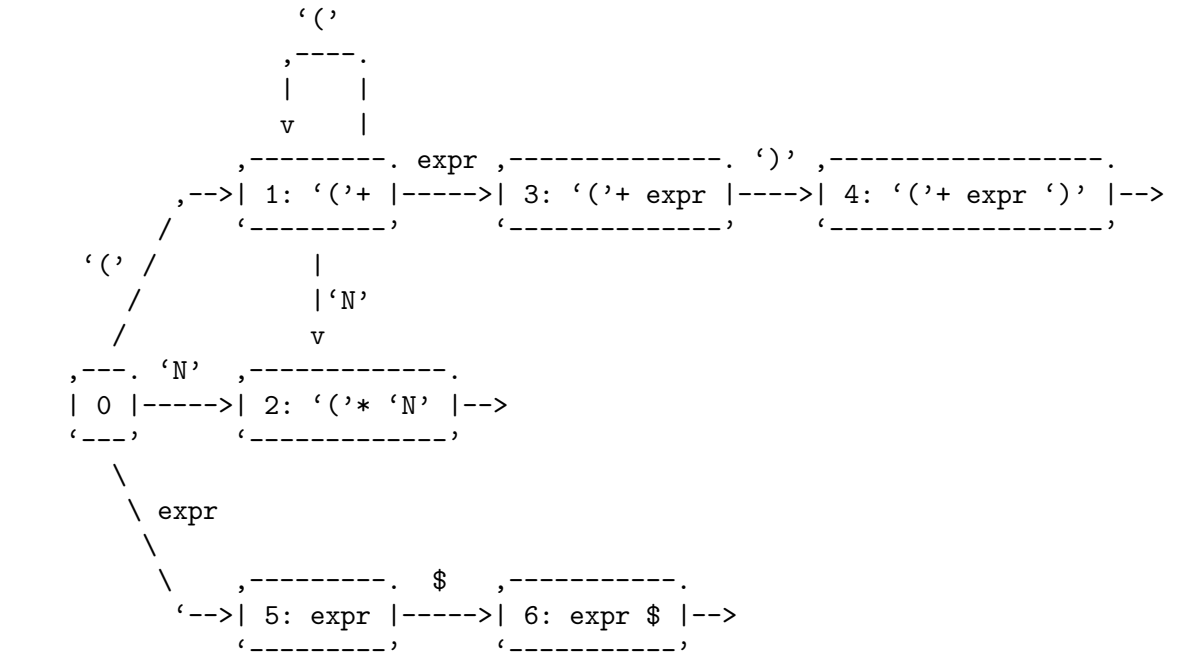
The tricky part is precisely knowing when to shift, and when to reduce: the strategy we followed consists in simply reducing as soon as we can, but how can we recognize stacks that can be reduced? To answer this question, we must also consider the cases where the input contains a syntax error, such as '`number number`'. Finally, our question amounts to "what are the valid stacks ready to be reduced". It is easy to see there are two cases: a possibly empty series of opening parentheses followed by either a lone `number`, or else by a '`(`', then an '`expr`', and a '`)`', or, finally a single '`expr`' and end of file.

Hey! "A possibly empty series of opening parentheses followed by...": this is typically what we used to denote with regular expressions! Here:

```
'('* ( expr $ | '(' expr ')' | number )
```

If we managed to describe our reducible stacks with regular expressions, it means we can recognize them using a finite state recognizer!

This FSR is easy to compute by hand, since its language is fairly small. It is depicted below, with an additional terminal symbol, '`$`', denoting the end of file, and '`N`' standing for `number`:



Example 7.2: *A Stack Recognizer for Balanced Expressions*

As expected, each final state denotes the recognition of a rule to reduce. For instance state 4 describes the completion of the first rule, '`expr: '(' expr ')'`', which we will denote as '`reduce 1`'. State 6 describes the acceptance of the whole text, which we will denote '`accept`'. Transitions labelled with tokens are shifts, for instance if the automaton is in state 1 and sees an '`N`', then it will '`shift 2`'. An transition labelled with a non terminal does not change the stack, for instance an '`expr`' in state 1 makes the automaton '`go to 4`'. Any impossible transition

(for instance receiving a closing parenthesis in state 0) denotes a syntax error: the text is not conform to the grammar.

In practice, because we want to produce efficient parsers, we won't restart the automaton from the beginning of the stack at each turn. Rather, we will remember the different states we reached. For instance the previous stack traces can be decorated with the states and actions as follows:

Step	Stack	Input	Action
1.	`0`	`( ( number ) ) $`	shift 1
2.	`0 ( 1`	`( number ) ) $`	shift 1
3.	`0 ( 1 ( 1`	`number ) ) $`	shift 2
4.	`0 ( 1 ( 1 number 2`	`) ) $`	reduce 2
5.	`0 ( 1 ( 1 expr`	`) ) $`	go to 3
6.	`0 ( 1 ( 1 expr 3`	`) ) $`	shift 4
7.	`0 ( 1 ( 1 expr 3 ) 4`	`) $`	reduce 1
8.	`0 ( 1 expr`	`) $`	go to 2
9.	`0 ( 1 expr 2`	`) $`	shift 4
10.	`0 ( 1 expr 2 ) 4`	`$`	reduce 1
11.	`0 expr`	`$`	go to 5
12.	`0 expr 5`	`$`	shift 6
13.	`0 expr 5 $ 6`		accept

Example 7.3: *Step by Step Analysis of* '`( ( number ) )`'

This technology is named LR(0) parsing, "L" standing for Left to Right Parsing, "R" standing for Rightmost Derivation[1], and "0" standing for no lookahead symbol: we never had to look at the input to decide whether to shift or to reduce.

Again, while the theory is simple, computing the valid stacks, designing an finite state stack recognizer which shifts or reduces, etc. is extremely tedious and error prone. A little of automation is most welcome.

7.2 Looking for Arithmetics

Except for a few insignificant details, the syntax introduced in the previous section is called BNF, standing for Backus-Naur form, from the name of its inventors: they used it to formalize the Algol 60 programming language. It is used to define *grammars*: a set of *rules* which describe the structure of a language, just as we used grammars to describe the English sentences at school. As for natural languages, two kinds of *symbols* are used to describe artificial languages: *terminal symbols* (e.g., '`he`', '`she`', etc. or '`+`', '`-`', etc.) and *nonterminal symbols* (e.g., "subject", or "operator"). Examples of grammars include

```
sentence:    subject predicate;
subject:     'she' | 'he' | 'it';
predicate:   verb noun-phrase | verb;
verb:        'eats';
noun-phrase: article noun | noun ;
article:     'the';
```

[1] When we reduce a rule, it is always at the top of our stack, corresponding to the rightmost part of the text input so forth. Some other parsing techniques, completely different, first handle the leftmost possible reductions.

```
noun:       'bananas' | 'coconuts';
```

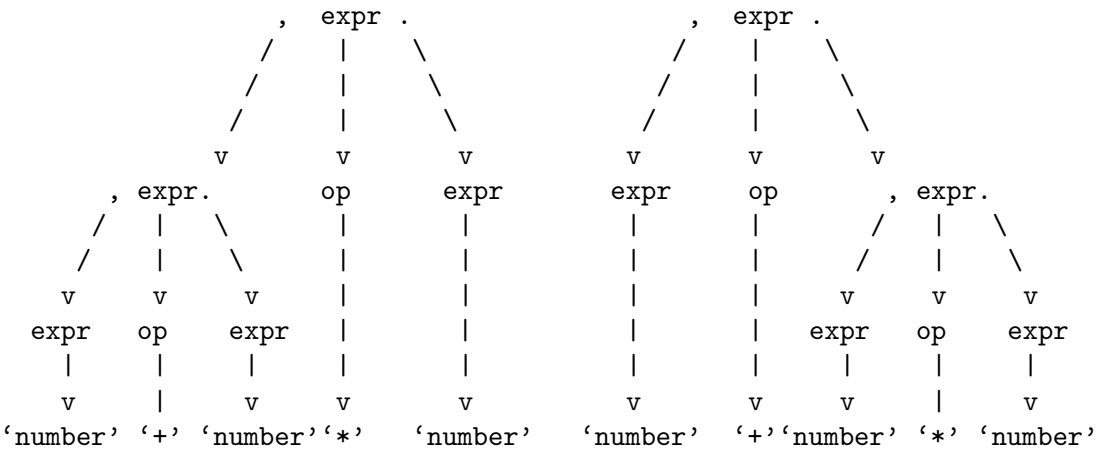
or

```
expr: expr op expr | '(' expr ')' | 'number';
op:   '+' | '-' | '*' | '/';
```

Example 7.4: *A Grammar for Arithmetics*

Such rules, which left hand side is always reduced to a single nonterminal, define so called *context free grammars*. Context free grammars are properly more powerful than regular expressions: any regular expression can be represented by a context free grammar, and there are context free grammars defining languages that regular expressions cannot denote (e.g., the nested parentheses).

Grammars can be *ambiguous*: some sentences can be understood in different ways. For instance the sentence '`number + number * number`' can be understood in two different ways by the grammar of the *example 7.4*:



Example 7.5: *Non Equivalent Parse Trees for* '`number + number * number`'

Because having different interpretations of a given sentence cannot be accepted for artificial languages (a satellite could easily be wasted if programmed in an ambiguous language), we will work with unambiguous grammars exclusively. For instance, we will have to refine the grammar of the *example 7.4* in order to use it with Bison.

Please note that even if limited to the minus operator, this grammar is still ambiguous: two parse trees represent '`number - number - number`'[2]. We, humans, don't find it ambiguous, because we know that by convention '`-`' is executed from left to right, or, in terms of parenthesis, it forms groups of two from left to right. This is called *left associativity*. There exist *right associative* operators, such as power, or assignment in C, that handle their rightmost parts first.

The following unambiguous grammar denotes the same language, but keeps only the conventional interpretation of subtraction: left associativity.

```
expr: expr '-' 'number' | 'number';
```

Example 7.6: *An Unambiguous Grammar for '-' Arithmetics*

Let's look at our stack recognizer again, on the input '`number - number`':

[2] These two parse trees are those of the *example 7.5*, with '`-`' replacing '`*`' and '`+`'.

Step	Stack	Input
1.		`number - number`
2.	`number`	`- number`
3.	`expr`	`- number`
4.	`expr -`	`number`
5.	`expr - number`	
6.	`expr`	

Example 7.7: *An LR(1) Parsing of* '`number - number`'

This time, we no longer can systematically apply the rule '`expr: 'number'`' each time we see a '`number`' on the top of the stack. In both step 2 and step 5, the top of the stack contains a `number` which can be reduced into an `expr`. We did reduce from step 2, but in step 5 we must not. If we did, our stack would then contain '`expr '-' expr`', out of which nothing can be done: all the '`number`'s *except* the last one, must be converted into an `expr`. We observe that looking at the next token, named the *lookahead*, solves the issue: if the top stack is '`number`', then if the lookahead is '`minus`', shift, if the lookahead is end-of-file, reduce the rule '`expr: 'num'`', and any other token is an error (think of '`number number`' for instance).

The theory we presented in the Section 7.1 [Looking for Balanced Expressions], page 157 can be extended to recognize patterns on the stack plus one lookahead symbol. This is called LR(1) parsing. As you will have surely guessed, LR(k) denotes this technique when peeking at the k next tokens in the input.

Unfortunately, even for reasonably simple grammars the automaton is quickly huge, so in practice, one limits herself to $k = 1$. Yet with a single lookahead, the automaton remains huge: a lot of research has been devoted to design reasonable limitations or compression techniques to have it fit into a reasonable machine. A successful limitation, which description falls out of the scope of this book, is known as LALR(1). A promising approach, DRLR, consists in recognizing the stack from its top instead of from its bottom. Intuitively there are less patterns to recognize since the pertinent information is usually near the top, hence the automaton is smaller.

In *A Discriminative Reverse Approach to LR(k) Parsing*, Fortes Gálvez José compares the handling of three grammars:

arithmetics — A small grammar similar to that of the *example 7.4*, but unambiguous. There are 5 terminals, 3 nonterminals, and 6 rules.

medium — A medium size grammar comprising 41 terminals, 38 nonterminals, and 93 rules.

programming — A real size grammar of a programming language, composed of 82 terminals, 68 nonterminals, and 234 rules.

by the three approaches. The size of the automaton is measured by its number of states and *entries* (roughly, its transitions):

Grammar	LR(1) States	LR(1) Entries	LALR(1) States	LALR(1) Entries	DRLR(1) States	DRLR(1) Entries
arithmetics	22	71	12	26	8	23
medium	773	6,874	173	520	118	781
programming	1,000+	15,000+	439	3,155	270	3,145

As you can see, LR(1) parsers are too expensive and as matter of fact there is no wide spread and used implementation, LALR(1) is reasonable and can be found in numerous tools such as Yacc and all its clones, and finally, because DRLR(1) addresses all the LR(1) grammars, it is an

appealing alternative to Yacc. Bison is an implementation of Yacc, hence it handles LALR(1) grammars, but might support DRLR(1) some day, providing the full expressive power of LR(1).

7.3 What is Bison

Yacc is a generator of efficient parsers. A *parser* is a program or routine which recognizes the structure of sentences. Yacc's input is composed of *rules* with associated actions. The rules must be *context free*, i.e., their left hand side is composed of a single nonterminal symbol, and their right hand side is composed of series of terminal and nonterminal symbols. When a rule is reduced, the associated C code is triggered.

Yacc is based on pushdown automata. It is a implementation of the LALR(1) parsing algorithm, which is sufficient for most programming languages, but can be too limited a framework to describe conveniently intricate languages.

Yacc, and all its declinations (CAMLYacc for CAML etc.) are used in numerous applications, especially compilers and interpreters. Hence its name: Yet Another Compiler Compiler.

Bison is a free software implementation of Yacc, as described by the POSIX standard. It provides a wide set of additional options and features, produces self contained portable code (no library is required), supports a more pleasant syntax, and stands as a standard of its own. Since most Yacc do have problems (inter-Yacc compatibility and actual bugs), all the reasons are in favor of using exclusively Bison: the portable C code it produces can be shipped in the package and will compile cleanly on the user's machine. It imposes no restriction on the license of the produced parser.

It is used by the GNU Compiler Collection for C, C++[3], the C preprocessor. Many other programming language tools use Bison or Yacc: GNU AWK, Perl, but it proves itself useful in reading structured files: a2ps uses it to parse its style sheets. It also helps decoding some limited forms of natural language: numerous GNU utilities use a Yacc grammar to decode dates such as '`2 days ago`', '`3 months 1 day`', '`25 Dec`', '`1970-01-01 00:00:01 UTC +5 hours`' etc.

GNU Gettext deserves a special mention with three different uses: one to parse its input files, another one to parse the special comments in these files, and one to evaluate the foreign language dependent rules defining the plural forms.

7.4 Bison as a Grammar Checker

Bison is dedicated to artificial languages, which, contrary to natural languages, must be absolutely unambiguous. More precisely it accepts only LALR(1) grammars, which formal definition amounts exactly to "parsable with a Bison parser". Although there are unambiguous grammars that are not LALR(1), we will henceforth use "ambiguous", or "insane", to mean "not parsable with Bison".

While the main purpose of Bison is to generate parsers, since writing good grammars is not an easy task, it proves itself very useful as a grammar checker (e.g., checking it is unambiguous). Again, a plain "error: grammar is insane" would be an information close to useless, fortunately Bison then provides means (i) to understand the insanity of a grammar, (ii) to solve typical ambiguities conveniently.

How exactly does an ambiguity reveal itself to Bison? We saw that Bison parsers are simple machines shifting tokens and reducing rules. As long as these machines know exactly what step to perform, the grammar is obviously sane. In other words, on an insane grammar it will

[3] There are plans to rewrite the C++ parser by hand because the syntax of the language is too intricate for LALR(1) parsing.

sometimes hesitate between different actions: should I shift, or should I reduce? And if I reduce, which rule should I reduce?

In Section 7.2 [Looking for Arithmetics], page 159, we demonstrated that the naive implementation arithmetics is ambiguous, even if reduced to subtraction. The following Bison file focuses on it. The mark '`%%`' is needed for Bison to know where the grammar starts:

```
%%
expr: expr '-' expr | "number";
```

Example 7.8: '`arith-1.y`' – *A Shift/Reduce Conflict*

Running `bison` on it, as expected, ends sadly:

```
$ bison arith-1.y
error arith-1.y contains 1 shift/reduce conflict.
```

As announced, ambiguities lead Bison to hesitate between several actions, here '`1 shift/reduce conflict`' stands for "there is a state from which I could either shift another token, or reduce a rule". But which? You may ask `bison` for additional details:

```
$ bison --verbose arith-1.y
error arith-1.y contains 1 shift/reduce conflict.
```

which will output the file '`arith-1.output`'[4]:

```
State 4 contains 1 shift/reduce conflict.

Grammar
    rule 0    $axiom -> expr $
    rule 1    expr   -> expr '-' expr
    rule 2    expr   -> "number"

Terminals, with rules where they appear
    $         (-1)
    '-'       (45) 1
    error    (256)
    "number" (257) 2

Nonterminals, with rules where they appear
    expr (5) on left: 1 2, on right: 1

State 0    $axion -> . expr $           (rule 0)
           expr   -> . expr '-' expr    (rule 1)
           expr   -> . "number"         (rule 2)
           "number"    shift, and go to state 1
           expr        go to state 2

State 1    expr  ->  "number" .         (rule 2)
           $default    reduce using rule 2 (expr)

State 2    $axiom -> expr . $           (rule 0)
           expr   -> expr . '-' expr    (rule 1)
           $           go to state 5
           '-'         shift, and go to state 3

State 3    expr -> expr '-' . expr      (rule 1)
           "number"    shift, and go to state 1
           expr        go to state 4
```

[4] In addition to minor formatting changes, this output has been modified. In particular the rule 0, which Bison hides, is made explicit.

```
State 4    expr -> expr . '-' expr      (rule 1)
           expr -> expr '-' expr .      (rule 1)
           '-'          shift, and go to state 3
           '-'          [reduce using rule 1 (expr)]
           $default     reduce using rule 1 (expr)

State 5    $axiom -> expr . $           (rule 0)
           $            shift, and go to state 6

State 6    $axiom -> expr $ .           (rule 0)
           $default     accept
```

You shouldn't be frightened by the contents: aside from being applied to a different grammar, this is nothing but the FSR we presented in *example 7.2*. Instead of being presented graphically, it is described by the list of its states and the actions that can be performed from each state. Another difference is that each state is represented by the degree of recognition of the various rules, instead of regular expressions.

For instance the state 3 is characterized by the fact that we recognized the first two symbols of '`expr '-' expr`', which is denoted '`expr -> expr '-' . expr`'. In that state, if we see a '`number`', we shift it, and go to state 1. If we see an '`expr`', we go to state 4. The reader is suggested to draw this automaton.

Bison draws our attention on the state 4, for "State 4 contains 1 shift/reduce conflict". Indeed, there are two possible actions when the lookahead, the next token in the input, is '`-`':

```
State 4    expr  ->  expr . '-' expr    (rule 1)
           expr  ->  expr '-' expr .    (rule 1)
           '-'          shift, and go to state 3
           '-'          [reduce using rule 1 (expr)]
           $default     reduce using rule 1 (expr)
```

Example 7.9: *State 4 contains 1 shift/reduce conflict*

We find again the ambiguity of our grammar: when reading '`number - number - number`', which leads to the stack being '`expr '-' expr`', when finding another '`-`', it doesn't know if it should:

– shift it ('`shift, and go to state 3`'), which results in

Stack	Input	
`0 expr 2 - 3 expr 4`	`- number $`	shift 3
`0 expr 2 - 3 expr 4 - 3`	`number $`	shift 1
`0 expr 2 - 3 expr 4 - 3 number 1`	`$`	reduce 2
`0 expr 2 - 3 expr 4 - 3 expr`	`$`	go to 4
`0 expr 2 - 3 expr 4 - 3 expr 4`	`$`	reduce 1
`0 expr 2 - 3 expr`	`$`	go to 4
`0 expr 2 - 3 expr 4`	`$`	reduce 1
`0 expr`	`$`	go to 5
`0 expr 5`	`$`	shift 6
`0 expr 5 $ 6`		accept

i.e., grouping the last two expressions, in other words understanding '`number - (number - number)`', or

– reduce rule 1 ('`[reduce using rule 1 (expr)]`'), which results in

`0 expr 2 - 3 expr 4`	`- number $`	reduce 1
`0 expr`	`- number $`	go to 2
`0 expr 2`	`- number $`	shift 3
`0 expr 2 - 3`	`number $`	shift 1

`0 expr 2 - 3 number 1`	`$`	reduce 2
`0 expr 2 - 3 expr`	`$`	goto 4
`0 expr 2 - 3 expr 4`	`$`	reduce 1
`0 expr`	`$`	go to 5
`0 expr 5`	`$`	shift 6
`0 expr 5 $ 6`		accept

i.e., grouping the first two expressions, in other words understanding '`(number - number) - number`'.

Well meaning, it even chose an alternative for us, shifting, which Bison signifies with the square brackets: possibility '`[reduce using rule 1 (expr)]`' will not be followed. Too bad, that's precisely not the choice we need...

The cure is well known: implement the grammar of the *example 7.6*. The reader is invited to implement it in Bison, and check the output file. In fact, the only difference between the two output files is that the state 4 is now:

```
State 4    expr  ->  expr '-' "number" .    (rule 1)
           $default    reduce using rule 1 (expr)
```

With some practice, you will soon be able to understand how to react to conflicts, spotting where they happen, and find another grammar which is unambiguous. Unfortunately...

7.5 Resolving Conflicts

Unfortunately, naive grammars are often ambiguous. The naive grammar for arithmetics, see *example 7.4*, is well-known for its ambiguities, unfortunately finding an unambiguous equivalent grammar is quite a delicate task and requires some expertise:

```
expr: expr '+' term
    | expr '-' term
    | term;
term: term '*' factor
    | term '/' factor
    | factor;
factor: '(' expr ')'
      | "number";
```

Example 7.10: *An Unambiguous Grammar for Arithmetics*

Worse yet: it makes the grammar more intricate to write. For instance, a famous ambiguity in the naive grammar for the C programming language is the "dangling `else`". Consider the following grammar excerpt:

```
%%
stmt: "if" "expr" stmt "else" stmt
    | "if" "expr" stmt
    | "stmt";
```

Example 7.11: '`ifelse-1.y`' – *Ambiguous grammar for* `if`/`else` *in C*

Although innocent looking, it is ambiguous and leads to two different interpretations of:

```
if   (expr-1)
if   (expr-2)
     statement-1
else statement-2
```

depending whether *statement-2* is bound to the first or second `if`. The C standard clearly mandates the second interpretation: a "dangling else" is bound to the closest `if`. How can one write a grammar for such a case? Again, the answer is not obvious, and requires some experience with grammars. It requires the distinction between "closed statements", i.e., those which `if` are saturated (they have their pending `else`), and non closed statements:

```
%%
stmt: closed_stmt
    | non_closed_stmt;
closed_stmt: "if" "expr" closed_stmt "else" closed_stmt
           | "stmt";
non_closed_stmt: "if" "expr" stmt
               | "if" "expr" closed_stmt "else" non_closed_stmt;
```

Example 7.12: *'`ifelse-2.y`' – Unambiguous Grammar for* `if`/`else` *in C*

And finally, since we introduce new rules, new symbols, our grammar gets bigger, hence the automaton gets fat, therefore it becomes less efficient (since modern processors cannot make full use of their caches with big tables).

Rather, let's have a closer look at *example 7.9*. There are two actions fighting to get into state 4: reducing the rule 1, or shifting the token '`-`'. We have a match between two opponents: if the rule wins, then '`-`' is right associative, if the token wins, then '`-`' is left associative.

What we would like to say is that "shifting '`-`' has priority over reducing '`expr: expr '-' expr`'". Bison is nice to us, and allows us to inform it of the associativity of the operator, it will handle the rest by itself:

```
%left '-'
%%
expr: expr '-' expr | "number";
```

Example 7.13: *'`arith-3.y`' – Using '`%left`' to Solve Shift/Reduce Conflicts*

```
$ bison --verbose arith-3.y
```

This is encouraging, but won't be enough for us to solve the ambiguity of the *example 7.5*:

```
%left '+'
%%
expr: expr '*' expr | expr '+' expr | "number";
```

Example 7.14: *'`arith-4.y`' – An Ambiguous Grammar for '`*`' vs. '`+`'*

```
$ bison --verbose arith-4.y
error arith-4.y contains 3 shift/reduce conflicts.
```

Diving into '`arith-4.output`' will help us understand the problem:

```
Conflict in state 5 between rule 2 and token '+' resolved as reduce.
State 5 contains 1 shift/reduce conflict.
State 6 contains 2 shift/reduce conflicts.
...
```

```
State 5     expr  ->  expr . '*' expr   (rule 1)
            expr  ->  expr '*' expr .   (rule 1)
            expr  ->  expr . '+' expr   (rule 2)
            '*'         shift, and go to state 3
            '*'         [reduce using rule 1 (expr)]
            $default    reduce using rule 1 (expr)

State 6     expr  ->  expr . '*' expr   (rule 1)
            expr  ->  expr '*' expr .   (rule 1)
            expr  ->  expr . '+' expr   (rule 2)
            '+'         shift, and go to state 3
            '*'         shift, and go to state 4
            '+'         [reduce using rule 1 (expr)]
            '*'         [reduce using rule 1 (expr)]
            $default    reduce using rule 1 (expr)
```

First note that it reported its understanding of '`%left '+'`': it is a match opposing token '`+`' against rule 2, and the rule is the winner.

State 6, without any surprise, has a shift/reduce conflict because we didn't define the associativity of '`*`', but states 5 and 6 have a new problem. In state 5, after having read '`expr + expr`', in front of a '`*`', should it shift it, or reduce '`expr + expr`'? Obviously it should reduce: the rule containing '`*`' should have precedence over the token '`+`'. Similarly, in state 6, the token '`*`' should have precedence over the rule containing '`+`'. Both cases can be summarized as "'`*`' has precedence over '`+`'".

Bison allows you to specify precedences simply by listing the '`%left`' and '`%right`' from the lowest precedence, to the highest. Some operators have equal precedence, for instance series of '`+`' and '`-`' must always be read from left to right: '`+`' and '`-`' have the same precedence. In this case, just put the two operators under the same '`%left`'.

Finally, we are now able to express our complete grammar of arithmetics:

```
%left '+' '-'
%left '*' '/'
%%
expr: expr '*' expr | expr '/' expr
    | expr '+' expr | expr '-' expr
    | '(' expr ')'
    | "number";
```

Example 7.15: '`arith-5.y`' – *Using Precedence to Solve Shift/Reduce Conflicts*

which is happily accepted by `bison`:

```
$ bison --verbose arith-5.y
```

The reader is invited to:

Implement the grammar of the *example 7.10*
: Read the output file, and see that indeed it requires a bigger automaton than the grammar of the *example 7.15*.

Play with the option '`--graph`'
: A complement of '`--verbose`' is '`--graph`', which outputs a VCG graph, to be viewed with `xvcg`. View the automata of the various grammars presented.

Explain the Failure of Factored Arithmetics
: Bison refuses the following grammar:

```
%left '+' '-'
%left '*' '/'
%%
expr: expr op expr | "number";
op: '+' | '-' | '*' | '/';
```

Example 7.16: *'arith-6.y' – Failing to Use Precedence*

```
$ bison arith-6.y
error arith-6.y contains 4 shift/reduce conflicts.
```

Can you explain why? Bear in mind the nature of the opponents in a shift/reduce match.

Solve the dangling `else`

The precedence of a rule is actually that of its *last* token. With this in mind, propose a simple implementation of the grammar of *example 7.10* in Bison.

We are now fully equipped to implement real parsers.

7.6 Simple Uses of Bison

Bison is a source generator, just as Gperf, Flex, and others. It takes a list of rules and actions, and produces a function triggering the action associated to the reduced rule. The input syntax allows for the traditional prologue, containing Bison directives and possibly some user declarations and initializations, and the epilogue, typically additional functions:

```
%{
  user-prologue
%}
bison-directives
%%
/* Comments. */
lfs:
    rhs-1 { action-1 }
  | rhs-2 { action-2 }
  | ...
  ;
...
%%
user-epilogue
```

Example 7.17: *Structure of a Bison Input File*

When run on a file 'foo.y', `bison` produces a file named 'foo.tab.c' and possibly 'foo.tab.h', 'foo.output', etc. This atrocious naming convention is a mongrel between the POSIX standards on the one hand requiring the output to be *always* named 'y.tab.c', and on the other hand the more logical 'foo.c'. You may set the output file name thanks to the `%output="`*parser-filename*`"` directive.

This file is a C source including, in addition to your *user-prologue* and *user-epilogue*, one function:

int yyparse () [Function]

Parse the whole stream of tokens delivered by successive calls to `yylex`. Trigger the *action* associated to the reduced rule.

Return 0 for success, and nonzero upon failures (such as parse errors).

You must provide `yylex`, but also `yyerror`, used to report parse errors.

For a start, we will implement the *example 7.1*. Because writing a scanner, which splits the input into tokens, is a job in itself, see Chapter 6 [Scanning with Gperf and Flex], page 127, we will simply rely on the command line arguments. In other words, we use the shell as a scanner (or rather, we let the user fight with its shells to explain where start and end the tokens).

```
%{ /* -*- C -*- */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <error.h>

static int yylex (void);
static void yyerror (const char *);
%}
%output="brackens-1.c"
%%
expr: '(' expr ')'
    | 'N'
    ;
%%
/* The current argument. */
static const char **args = NULL;

static int
yylex (void)
{
  /* No token stands for end of file. */
  if (!*++args)
    return 0;
  /* Parens stand for themselves.  'N' denotes a number. */
  if (strlen (*args) == 1 && strchr ("()N", **args))
    return **args;
  /* An error. */
  error (EXIT_FAILURE, 0, "invalid argument: %s", *args);
  /* Pacify GCC that knows ERROR may return. */
  return -1;
}

void
yyerror (const char *msg)
{
  error (EXIT_FAILURE, 0, "%s", msg);
}

int
main (int argc, const char *argv[])
{
  args = argv;
  return yyparse ();
}
```

Example 7.18: ‘`brackens-1.y`’ – *Nested Parentheses Checker*

Which we compile and run:

```
$ bison brackens-1.y
$ gcc -Wall brackens-1.c -o brackens-1
$ ./brackens-1 '(' 'N' ')'
$ ./brackens-1 '(' 'n' ')'
error ./brackens-1: invalid argument: n
$ ./brackens-1 '(' '(' 'N' ')' ')'
$ ./brackens-1 '(' '(' 'N' ')' ')' ')'
error ./brackens-1: parse error
```

It works quite well! Unfortunately, when given an invalid input is it quite hard to find out where the actual error is. Fortunately you can ask Bison parsers to provide more precise information by defining `YYERROR_VERBOSE` to 1. You can also use the Bison directive `%debug` which introduces a variable, `yydebug`, to make your parser verbose:

```
%{ /* -*- C -*- */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <error.h>

static int yylex (void);
static void yyyerror (const char *msg);
#define YYERROR_VERBOSE 1
%}
%debug
%output="brackens-2.c"
%%
expr: '(' expr ')'
    | 'N'
    ;
%%

int
main (int argc, const char *argv[])
{
  yydebug = getenv ("YYDEBUG") ? 1 : 0;
  args = argv;
  return yyparse ();
}
```

Example 7.19: ‘`brackens-2.y`’ – *Nested Parentheses Checker*

which, when run, produces:

```
$ ./brackens-2 '(' '(' 'N' ')' ')' ')'
./brackens-2: parse error, unexpected ')', expecting $
```

Wow! The error message is *much* better! But it is still not clear which one of the arguments is provoking the error. Asking traces from the parser gives some indication:

```
$ YYDEBUG=1 ./brackens-2 '(' '(' 'N' ')' ')' ')'
Starting parse
```

```
Entering state 0
Reading a token: Next token is 40 ('(')
Shifting token 40 ('('), Entering state 1
Reading a token: Next token is 40 ('(')
Shifting token 40 ('('), Entering state 1
Reading a token: Next token is 78 ('N')
Shifting token 78 ('N'), Entering state 2
Reducing via rule 2 (line 32), 'N'  -> expr
state stack now 0 1 1
Entering state 3
Reading a token: Next token is 41 (')')
Shifting token 41 (')'), Entering state 4
Reducing via rule 1 (line 32), '(' expr ')'  -> expr
state stack now 0 1
Entering state 3
Reading a token: Next token is 41 (')')
Shifting token 41 (')'), Entering state 4
Reducing via rule 1 (line 32), '(' expr ')'  -> expr
state stack now 0
Entering state 5
Reading a token: Next token is 41 (')')
error ./brackens-2: parse error, unexpected ')', expecting $
```

The reader is invited to compare these traces with the "execution" by hand of the *example 7.3*: it is just the same!

Two things are worth remembering. First, always try to produce precise error messages, since once an error is diagnosed, the user still has to locate it in order to fix it. I, for one, don't like tools that merely report the line where the error occurred, because often *several* very similar expressions within that line can be responsible for the error. In that case, I often split my line into severals, recompile, find the culprit, educate it, and join again these lines... And second, never write dummy `printf` to check that your parser works: it is a pure waste of time, since Bison does this on command, and produces extremely readable traces. Take some time to read the example above, and in particular spot on the one hand side the tokens and the shifts and on the other hand the reductions and the rules. But forget about "states", for the time being.

7.7 Using Actions

Of course real applications need to process the input, not merely to validate it as in the previous section. For instance, when processing arithmetics, one wants to compute the result, not just say "this is a valid/invalid arithmetical expression".

We stem on several problems. First of all, up to now we managed to use single characters as tokens: the code of the character is used as token number. But all the integers must be mapped to a single token type, say `INTEGER`.

Bison provide the `%token` directive to declare token types:

```
%token INTEGER FLOAT STRING
```

Secondly, tokens such as numbers have a value. To this end, Bison provides a global variable, `yylval`, which the scanner must fill with the appropriate value. But imagine our application also had to deal with strings, or floats etc.: we need to be able to specify several value types, and associate a value type to a token type.

To declare the different value types to Bison, use the `%union` directive. For instance:

```
%union
{
  int   ival;
  float fval;
  char *sval;
}
```

This results in the definition of the type "token value": `yystype`[5]. The scanner needs the token types and token value types: if given '`--defines`' `bison` creates '`foo.h`' which contains their definition. Alternatively, you may use the `%defines` directive.

Then map token types to value types:

```
%token <ival> INTEGER
%token <fval> FLOAT
%token <sval> STRING
```

But if tokens have a value, then so have some nonterminals! For instance, if `iexpr` denotes an integer expression, then we must also specify that (i) it has a value, (ii) of type `INTEGER`. To this end, use `%type`:

```
%type <ival> iexpr
%type <fval> fexpr
%type <sval> sexpr
```

We already emphasized the importance of traces: together with the report produced thanks to the option '`--verbose`', they are your best partners to debug a parser. Bison lets you improve the traces of tokens, for instance to output token values in addition to token types. To use this feature, just define the following macro:

YYPRINT (*File, Type, Value*) [Macro]
Output on *File* a description of the *Value* of a token of *Type*.

For various technical reasons, I suggest the following pattern of use:

```
%{
...
#define YYPRINT(File, Type, Value) yyprint (File, Type, &Value)
static void yyprint (FILE *file, int type, const yystype *value);
...
%}
%%
...
%%
static void
yyprint (FILE *file, int type, const yystype *value)
{
  switch (type)
    {
    case INTEGER:
      fprintf (file, " = %d", value->ival);
      break;
    case FLOAT:
      fprintf (file, " = %f", value->fval);
      break;
```

[5] Because of the compatibility with POSIX Yacc, the type `YYSTYPE` is also defined. For sake of inter Yacc portability, use the latter only. But for sake of your mental balance, use Bison only.

```
      case STRING:
        fprintf (file, " = \"%s\"", value->sval);
        break;
      }
  }
```

The most important difference between a mere validation of the input and an actual computation, is that we must be able to equip the parser with actions. For instance, once we have recognized that '`1 + 2`' is a '`INTEGER + INTEGER`', we must be able to (i) compute the sum, which requires fetching the value of the first and third tokens, and (ii) "return" 3 as the result.

To associate an action with a rule, just put the action after the rule, between braces. To access the value of the various symbols of the right hand side of a rule, Bison provides pseudo-variables: '`$1`' is the value of the first symbol, '`$2`' is the second etc. Finally to "return" a value, or rather, to set the value of the left hand side symbol of the rule, use '`$$`'.

Therefore, a calculator typically includes:

```
iexpr: iexpr '+' iexpr { $$ = $1 + $3 }
     | iexpr '-' iexpr { $$ = $1 - $3 }
     | iexpr '*' iexpr { $$ = $1 * $3 }
     | iexpr '/' iexpr { $$ = $1 / $3 }
     | INTEGER         { $$ = $1 }
     ;
```

Please, note that we used '`$3`' to denote the value of the third symbol, even though the second, the operator, has no value.

Putting this all together leads to the following implementation of the *example 7.15*:

```
%{ /* -*- C -*- */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <error.h>
#include "calc.h"

#define YYERROR_VERBOSE 1
#define yyerror(Msg) error (EXIT_FAILURE, 0, "%s", Msg)

#define YYPRINT(File, Type, Value) yyprint (File, Type, &Value)
static void yyprint (FILE *file, int type, const yystype *value);

static int yylex (void);
%}
%debug
%defines
%output="calc.c"
%union
{
  int ival;
}
%token <ival> NUMBER
%type  <ival> expr input
```

```
%left '+' '-'
%left '*' '/'
%%
input: expr         { printf ("%d\n", $1) }
     ;

expr: expr '*' expr { $$ = $1 * $3 }
    | expr '/' expr { $$ = $1 / $3 }
    | expr '+' expr { $$ = $1 + $3 }
    | expr '-' expr { $$ = $1 - $3 }
    | '(' expr ')'  { $$ = $2 }
    | NUMBER        { $$ = $1 }
    ;
%%
/* The current argument. */
static const char **args = NULL;

static void
yyprint (FILE *file, int type, const yystype *value)
{
  switch (type)
    {
    case NUMBER:
      fprintf (file, " = %d", value->ival);
      break;
    }
}

static int
yylex (void)
{
  /* No token stands for end of file. */
  if (!*++args)
    return 0;
  /* Parens and operators stand for themselves. */
  if (strlen (*args) == 1 && strchr ("()+-*/", **args))
    return **args;
  /* Integers have a value. */
  if (strspn (*args, "0123456789") == strlen (*args))
    {
      yylval.ival = strtol (*args, NULL, 10);
      return NUMBER;
    }
  /* An error. */
  error (EXIT_FAILURE, 0, "invalid argument: %s", *args);
  /* Pacify GCC which knows ERROR may return. */
  return -1;
}
```

```
int
main (int argc, const char *argv[])
{
  yydebug = getenv ("YYDEBUG") ? 1 : 0;
  args = argv;
  return yyparse ();
}
```

Example 7.20: *'calc.y' – A Simple Integer Calculator*

Its execution is satisfying:

```
$ bison calc.y
$ gcc calc.c -o calc
$ ./calc 1 + 2 \* 3
7
$ ./calc 1 + 2 \* 3 \*
error ./calc: parse error, unexpected $, expecting NUMBER or '('
$ YYDEBUG=1 ./calc 51
Starting parse
Entering state 0
Reading a token: Next token is 257 (NUMBER = 51)
Shifting token 257 (NUMBER), Entering state 1
Reducing via rule 7 (line 56), NUMBER  -> expr
state stack now 0
Entering state 3
Reading a token: Now at end of input.
Reducing via rule 1 (line 48), expr  -> input
51
state stack now 0
Entering state 14
Now at end of input.
Shifting token 0 ($), Entering state 15
Now at end of input.
```

well, almost

```
$ ./calc 1 / 0
floating point exception  ./calc 1 / 0
```

7.8 Advanced Use of Bison

The example of the previous section is very representative of real uses of Bison, except for its scale. Nevertheless, there remains a few issues to learn before your Bison expert graduation. In this section, we complete a real case example started in Section 6.2.6 [Advanced Use of Flex], page 151: a reimplementation of M4's `eval` builtin, using Flex and Bison.

Bison parsers are often used on top of Flex scanners. As already explained, the scanner then needs to know the definition of the token types, and the token value types: to this end use '`--defines`' or `%defines` to produce a header containing their definition.

The real added value of good parsers over correct parsers is in the handling of errors: the accuracy and readability of the error messages, and their ability to proceed as much as possible instead of merely dying at the first glitch[6].

[6] No one would ever accept a compiler which reports only the first error it finds, and then exits!

It is an absolute necessity for error messages to specify the precise location of the errors. To this end, when given '`--locations`' or `%locations`, `bison` provides an additional type, `yyltype` which defaults to:

```
typedef struct yyltype
{
  int first_line;
  int first_column;

  int last_line;
  int last_column;
} yyltype;
```

and another global variable, `yylloc`, which the scanner is expected to set for each token (see Section 6.2.6 [Advanced Use of Flex], page 151). Then, the parser will automatically keep track of the locations not only of the tokens, but also of nonterminals. For instance, it knows that the location of '`1 + 20 * 300`' starts where '`1`' starts, and ends where '`300`' does. As with '`$$`', '`$1`' etc., you may use '`@$`', '`@1`' to set/access the location of a symbol, for instance in "division by 0" error messages.

It is unfortunate, but the simple result of history, that improving the error messages requires some black incantations:

```
#define YYERROR_VERBOSE 1
#define yyerror(Msg) yyerror (&yylloc, Msg)
```

in other words, even with `%locations` Bison does not pass the location to `yyerror` by default. Similarly, enriching the traces with the locations requires:

```
/* FIXME: There used to be locations here.  */
#define YYPRINT(File, Type, Value) yyprint (File, /* &yylloc, */ Type, &Value)
```

In the real world, using a fixed set of names such as `yylex`, `yyparse`, `yydebug` and so on is wrong: you may have several parsers in the same program, or, in the case of libraries, you must stay within your own name space. Similarly, using global variables such as `yylval`, `yylloc` is dangerous, and prevents any recursive use of the parser.

These two problems can be solved with Bison: use '`%name-prefix="`*prefix*`"`' to rename of the '`yyfoo`'s into '*prefix*`foo`'s, and use `%pure-parser` to make Bison define `yylval` etc. as variables of `yyparse` instead of global variables.

If you are looking for global-free parsers, then, obviously, you need to be able to exchange information with `yyparse`, otherwise, how could it return something to you! Bison provides an optional user parameter, which is typically a structure in which you include all the information you might need. This structure is conventionally called a *parser control structure*, and for consistency I suggest naming it `yycontrol`. To define it, merely define `YYPARSE_PARAM`:

```
#define YYPARSE_PARAM yycontrol
```

Moving away from '`yy`' via `%name-prefix`, and from global variables via `%pure-parser` also deeply change the protocol between the scanner and the parser: before, `yylex` expected to find a global variable named `yylval`, but now it is named `foolval` and it is not global any more!

Scanners written with Flex can easily be adjusted: give it '`%option prefix="`*prefix*`"`' to change the '`yy`' prefix, and explain, (i) to Bison that you want to pass an additional parameter to `yylex`:

```
#define YYLEX_PARAM yycontrol
```

and (ii) on the other hand, to Flex that the prototype of `yylex` is to be changed:

```
#define YY_DECL \
```

```
int yylex (yystype *yylval, yyltype *yylloc, yycontrol_t *yycontrol)
```

Putting this all together for our `eval` reimplementation gives:

```
%debug
%defines
%locations
%pure-parser
%name-prefix="yleval_"
/* Request detailed parse error messages. */
%error-verbose

%{
#if HAVE_CONFIG_H
#  include <config.h>
#endif

#include <m4module.h>
#include "yleval.h"

/* When debugging our pure parser, we want to see values and locations
   of the tokens.  */
/* FIXME: Locations. */
#define YYPRINT(File, Type, Value) \
        yyprint (File, /* FIXME: &yylloc, */ Type, &Value)
static void yyprint (FILE *file, /* FIXME: const yyltype *loc, */
                     int type, const yystype *value);
/*
void yyerror (const YYLTYPE *location, yleval_control_t *yleval_control,
              const char *message, ...);
*/

%}

/* Pass the control structure to YYPARSE and YYLEX. */
%parse-param "yleval_control_t *yycontrol", "yycontrol"
%lex-param   "yleval_control_t *yycontrol", "yycontrol"

/* Only NUMBERS have a value.  */
%union
{
  number number;
};
```

The name of the tokens is an implementation detail: the user has no reason to know `NUMBER` is your token type for numbers. Unfortunately, these token names appear in the verbose error messages, and we want them! For instance, in the previous section, we implemented `calc`:

```
$ ./calc '(' 2 + ')'
error ./calc: parse error, unexpected ')', expecting NUMBER or '('
```

Bison lets you specify the "visible" symbol names to produce:

```
$ ./calc '(' 2 + ')'
error ./calc: parse error, unexpected ')', expecting "number" or '('
```

which is not perfect, but still better. In our case:

```
/* Define the tokens together with there human representation. */
%token YLEVAL_EOF 0 "end of string"
%token <number> NUMBER "number"
%token LEFTP "(" RIGHTP ")"
%token LOR  "||"
%token LAND "&&"
%token OR   "|"
%token XOR  "^"
%token AND  "&"
%token EQ "=" NOTEQ "!="
%token GT ">" GTEQ ">=" LS "<" LSEQ "<="
%token LSHIFT "<<" RSHIFT ">>"
%token PLUS "+" MINUS "-"
%token TIMES "*" DIVIDE "/" MODULO "%" RATIO ":"
%token EXPONENT "**"
%token LNOT "!" NOT "~" UNARY

%type <number> exp
```

There remains to define the precedences for all our operators, which ends our prologue:

```
/* Specify associativities and precedences from the lowest to the
   highest.  */
%left LOR
%left LAND
%left OR
%left XOR
%left AND
/* These operators are non associative, in other words, we forbid
   '-1 < 0 < 1'.  C allows this, but understands it as
   '(-1 < 0) < 1' which evaluates to... false.  */
%nonassoc EQ NOTEQ
%nonassoc GT GTEQ LS LSEQ
%nonassoc LSHIFT RSHIFT
%left  PLUS MINUS
%left  TIMES DIVIDE MODULO RATIO
%right EXPONENT
/* UNARY is used only as a precedence place holder for the
   unary plus/minus rules.  */
%right LNOT NOT UNARY
%%
```

The next section of '`ylparse.y`' is the core of the grammar: its rules. The very first rule deserves special attention:

```
result: { LOCATION_RESET (yylloc) } exp { yycontrol->result = $2; };
```

it aims at initializing `yylloc` before the parsing actually starts. It does so via a so called *mid-rule action*, i.e., a rule which is executed before the whole rule is recognized[7]. Then it looks

[7] The attentive reader will notice that the notion of mid-rule action does not fit with LALR(1) techniques: an action can only be performed once a whole rule is recognized. The paradox is simple to solve: internally Bison rewrites the above fragment as

```
result: @1 exp    { yycontrol->result = $2; };
@1: /* Empty. */  { LOCATION_RESET (yylloc); };
```

where '`@1`' is a fresh nonterminal. You are invited to read the '`--verbose`' report which does include these invisible symbols and rules.

for a single expression, which value consists in the result of the parse: we store it in the parser control structure.

The following chunk addresses the token `NUMBER`. It relies on the default action: '`$$ = $1`'.

```
/* Plain numbers. */
exp:
  NUMBER
;

/* Parentheses. */
exp:
  LEFTP exp RIGHTP { $$ = $2; }
;
```

The next bit of the grammar describes arithmetics. Two treatments deserve attention:

Unary Operators
: We want the unary minus and plus to bind extremely tightly: their precedence is higher than that binary plus and minus. But, as already explained, the precedence of a rule is that of its last token... by default... To override this default, we use `%prec` which you can put anywhere in the rule. The rules below read as "the rules for unary minus and plus have the precedence of the precedence place holder `UNARY`".

Semantic Errors
: Not all the exponentiations are valid ('`2 ^ -1`'), nor are all the divisions and moduli ('`1 / 0`', '`1 % 0`'). When such errors are detected, we abort the parsing by invoking `YYABORT`.

```
/* Arithmetics. */
exp:
  PLUS  exp        { $$ =   $2; }   %prec UNARY
| MINUS exp        { $$ = - $2; }   %prec UNARY
| exp PLUS     exp { $$ = $1 + $3; }
| exp MINUS    exp { $$ = $1 - $3; }
| exp TIMES    exp { $$ = $1 * $3; }
| exp DIVIDE   exp { if (!yldiv (yycontrol, &@$, &$$, $1, $3)) YYABORT; }
| exp MODULO   exp { if (!ylmod (yycontrol, &@$, &$$, $1, $3)) YYABORT; }
| exp RATIO    exp { if (!yldiv (yycontrol, &@$, &$$, $1, $3)) YYABORT; }
| exp EXPONENT exp { if (!ylpow (yycontrol, &@$, &$$, $1, $3)) YYABORT; }
;

/* Booleans. */
exp:
  LNOT exp     { $$ = ! $2; }
| exp LAND exp { $$ = $1 && $3; }
| exp LOR  exp { $$ = $1 || $3; }
;

/* Comparisons. */
exp:
  exp EQ    exp { $$ = $1 == $3; }
| exp NOTEQ exp { $$ = $1 != $3; }
| exp LS    exp { $$ = $1 <  $3; }
| exp LSEQ  exp { $$ = $1 <= $3; }
| exp GT    exp { $$ = $1 >  $3; }
| exp GTEQ  exp { $$ = $1 >= $3; }
;
```

```
/* Bitwise. */
exp:
  NOT exp        { $$ = ~ $2; }
| exp AND    exp { $$ = $1 & $3; }
| exp OR     exp { $$ = $1 | $3; }
| exp LSHIFT exp { $$ = $1 << $3; }
| exp RSHIFT exp { $$ = $1 >> $3; }
;
%%
```

Finally, the epilogue of our parser consists in the definition of the tracing function, `yyprint`:

```
/*---------------------------------------------------------------.
| When debugging the parser, display tokens' locations and values.  |
`---------------------------------------------------------------*/

static void
yyprint (FILE *file,
         /* FIXME: const yyltype *loc, */ int type, const yystype *value)
{
  fputs (" (", file);
  /* FIXME: LOCATION_PRINT (file, *loc); */
  fputs (")", file);
  switch (type)
    {
    case NUMBER:
      fprintf (file, " = %ld", value->number);
      break;
    }
}
```

7.9 The `yleval` Module

Our job is almost done, there only remains to create the leader: the M4 module which receives the expression from M4, launches the parser on it, and return the result or an error to M4. The most relevant bits of this file, '`yleval.c`', are included below. The reader is also referred to '`yleval.h`' and '`ylscan.l`', see Section 6.2.6 [Advanced Use of Flex], page 151.

```
/*-------------------------------------------------------------------.
| Record an error occurring at location LOC and described by MSG.  |
`-------------------------------------------------------------------*/

void
yleval_error (const yyltype *loc, yleval_control_t *control,
              const char *msg, ...)
{
  va_list ap;
  /* Separate different error messages with a new line. */
  fflush (control->err);
  if (control->err_size)
    putc ('\n', control->err);
  LOCATION_PRINT (control->err, *loc);
  fprintf (control->err, ": ");
  va_start (ap, msg);
  vfprintf (control->err, msg, ap);
  va_end (ap);
}
```

This first function, `yleval_error`, is used to report the errors. Since there can be several errors in a single expression, it uses a facility provided by the GNU C Library: pseudo streams which are in fact hooked onto plain `char *` buffers, grown on demand. We will see below how to initialize this stream, `err`, part of our control structure defined as:

```
typedef struct yleval_control_s
{
  /* To store the result. */
  number result;

  /* A string stream.  */
  FILE *err;
  char *err_str;
  size_t err_size;
} yleval_control_t;
```

The following function computes the division, and performs some semantic checking. `ylmod` and `ylpow` are similar.

```
/*-------------------------------------------------------------------.
| Compute NUMERATOR / DENOMINATOR, and store in RESULT.  On errors, |
| produce an error message, and return FALSE.                       |
`-------------------------------------------------------------------*/

boolean
yldiv (yleval_control_t *control, const yyltype *loc,
       number *result, number numerator, number denominator)
{
  if (!denominator)
    {
      yleval_error (loc, control, "Division by zero");
      *result = 0;
      return FALSE;
    }

  *result = numerator / denominator;
  return TRUE;
}
```

Now, the conductor of the whole band, the builtin `yleval` is defined as follows. Its first part is dedicated to initialization, and to decoding of the optional arguments passed to the builtin:

```
/*-----------------------------------.
| yleval(EXPRESSION, [RADIX], [MIN]) |
`-----------------------------------*/

M4BUILTIN_HANDLER (yleval)
{
  int radix = 10;
  int min = 1;

  /* Initialize the parser control structure, in particular
     open the string stream ERR. */
  yleval_control_t yleval_control;
  yleval_control.err =
    open_memstream (&yleval_control.err_str,
                    &yleval_control.err_size);

  /* Decode RADIX, reject invalid values. */
  if (argc >= 3 && !m4_numeric_arg (argc, argv, 2, &radix))
    return;

  if (radix <= 1 || radix > 36)
    {
      M4ERROR ((warning_status, 0,
                _("Warning: %s: radix out of range: %d"),
                M4ARG(0), radix));
      return;
    }
```

```
  /* Decode MIN, reject invalid values. */
  if (argc >= 4 && !m4_numeric_arg (argc, argv, 3, &min))
    return;

  if (min <= 0)
    {
      M4ERROR ((warning_status, 0,
                _("Warning: %s: negative width: %d"),
                M4ARG(0), min));
      return;
    }
```

Then it parses the expression, and outputs it result.

```
  /* Feed the scanner with the EXPRESSION. */
  yleval__scan_string (M4ARG (1));
  /* Launch the parser. */
  yleval_parse (&yleval_control);

  /* End the ERR stream.  If it is empty, the parsing is
     successful and we return the value, otherwise, we report
     the error messages. */
  fclose (yleval_control.err);
  if (!yleval_control.err_size)
    {
      numb_obstack (obs, yleval_control.result, radix, min);
    }
  else
    {
      M4ERROR ((warning_status, 0,
                _("Warning: %s: %s: %s"),
                M4ARG (0), yleval_control.err_str, M4ARG (1)));
      free (yleval_control.err_str);
    }
}
```

Example 7.21: '`yleval.y`' – *Builtin* `yleval` *(continued)*

It is high time to play with our module! Here are a few sample runs:

```
$ echo "yleval(1)" | m4 -M . -m yleval
1
```

Good news, we seem to agree on the definition of 1,

```
$ echo "yleval(1 + 2 * 3)" | m4 -M . -m yleval
7
```

and on the precedence of multiplication over addition. Let's exercise big numbers, and the radix:

```
$ echo "yleval(2 ** 2 ** 2 ** 2 - 1)" | m4 -M . -m yleval
65535
$ echo "yleval(2 ** 2 ** 2 ** 2 - 1, 2)" | m4 -M . -m yleval
1111111111111111
```

How about tickling the parser with a few parse errors:

```
$ echo "yleval(2 *** 2)" | m4 -M . -m yleval
error m4: stdin: 1: Warning: yleval: 1.5: parse error, unexpected "*": 2 *** 2
```

Wow! Now, *that*'s what I call an error message: the fifth character, in other words the third '`*`', should not be there. Nevertheless, at this point you may wonder how the stars were grouped[8]. Because you never know what the future is made of, I strongly suggest that you *always* equip your scanners and parsers with runtime switches to enable/disable tracing. This is the point of the following additional builtin, `yldebugmode`:

```
/*-----------------------------.
| yldebugmode([REQUEST1], ...) |
`-----------------------------*/

M4BUILTIN_HANDLER (yldebugmode)
{
  /* Without arguments, return the current debug mode. */
  if (argc == 1)
    {
      m4_shipout_string (obs,
                         yleval__flex_debug ? "+scanner" : "-scanner", 0, TRUE);
      obstack_1grow (obs, ',');
      m4_shipout_string (obs,
                         yleval_debug ? "+parser" : "-parser", 0, TRUE);
    }
  else
    {
      int arg;
      for (arg = 1; arg < argc; ++arg)
        if (!strcmp (M4ARG (arg), "+scanner"))
          yleval__flex_debug = 1;
        else if (!strcmp (M4ARG (arg), "-scanner"))
          yleval__flex_debug = 0;
        else if (!strcmp (M4ARG (arg), "+parser"))
          yleval_debug = 1;
        else if (!strcmp (M4ARG (arg), "-parser"))
          yleval_debug = 0;
        else
          M4ERROR ((warning_status, 0,
                    _("%s: invalid debug flags: %s"),
                   M4ARG (0), M4ARG (arg)));
    }
}
```

Applied to our previous example, it clearly demonstrates how '`2 *** 2`' is parsed[9]:

```
$ echo "yldebugmode(+parser)yleval(2 *** 2)" | m4 -M . -m yleval
error Starting parse
error Entering state 0
error Reducing via rule 1 (line 100),  -> @1
error state stack now 0
error Entering state 2
error Reading a token: Next token is 257 ("number" (1.1) = 2)
```

[8] Actually you should already know that Flex chose the longest match first, therefore it returned '`**`' and then '`*`'. See Section 6.2.1 [Looking for Tokens], page 141.

[9] The same attentive reader who was shocked by the concept of mid-rule actions, see Section 7.8 [Advanced Use of Bison], page 175, will notice the reduction of the invisible '`@1`' symbol below.

```
error Shifting token 257 ("number"), Entering state 4
error Reducing via rule 3 (line 102), "number"  -> exp
error state stack now 0 2
error Entering state 10
error Reading a token: Next token is 279 ("**" (1.3-4))
error Shifting token 279 ("**"), Entering state 34
error Reading a token: Next token is 275 ("*" (1.5))
error Error: state stack now 0 2 10
error Error: state stack now 0 2
error Error: state stack now 0
error m4: stdin: 1: Warning: yleval: 1.5: parse error, unexpected "*": 2 *** 2
```

which does confirm '`***`' is split into '`**`' and then '`*`'.

7.10 Using Bison with the GNU Build System

Autoconf and Automake provide some generic Yacc support, but no Bison dedicated support. In particular Automake expects '`y.tab.c`' and '`y.tab.h`' as output files, which makes it go through various hoops to prevent several concurrent invocations of `yacc` to overwrite each others' output.

As Gperf, Lex, and Flex, Yacc and Bison are maintainer requirements: someone changing the package needs them, but since their result is shipped, a regular user does not need them. Really, don't bother with different Yaccs, Bison alone will satisfy all your needs, and if it doesn't, just improve it!

If you use Autoconf's `AC_PROG_YACC`, then if `bison` is found, it sets `YACC` to '`bison --yacc`', otherwise to '`byacc`' if it exist, otherwise to '`yacc`'. It seems so much simpler to simply set `YACC` to '`bison`'! But then you lose the assistance of Automake, and in particular of `missing` (see Section 6.1.6 [Using Gperf with the GNU Build System], page 140). So I suggest simply sticking to `AC_PROG_YACC`, and let Automake handle the rest.

Then, in your '`Makefile.am`', merely list Bison sources as ordinary source files:

```
LDFLAGS = -no-undefined

pkglibexec_LTLIBRARIES = yleval.la
yleval_la_SOURCES = ylparse.y ylscan.l yleval.c yleval.h ylparse.h
yleval_la_LDFLAGS = -module
```

and that's all.

7.11 Exercises on Bison

Error Recovery

> There is one single important feature of Yacc parsers that we have not revealed here: the use of the `error` token to recover from errors. The general idea is that when the parser finds an error, it tries to find the nearest rule which claims to be ready to return to normal processing after having thrown away embarrassing symbols. For instance:
>
> ```
> exp: '(' error ')'
> ```
>
> But including recovery means you must pretend everything went right. In particular, an `exp` is expected to have a value, this rule *must* provide a valid `$$`. In our case, the

absence of a rule means proceeding with a random integer, but in other applications it may be a wandering pointer!

Equip `yleval` with proper error recovery. See section "Writing rules for error recovery" in *Bison – The YACC-compatible Parser Generator*, for detailed explanations on `error` and `yyerrok`.

LR(2) Grammars

Consider the following excerpt of the grammar of Bison grammars:

```
grammar: rule
       | grammar rule
rule: symbol ':' ride-hand-side
right-hand-side: /* Nothing */
               | right-hand-side symbol
symbol: 's'
```

Convince yourself that Bison cannot accept this grammar either by (i) feeding it to `bison` and understanding the conflict, (ii) drawing the LR(1) automaton and understanding the conflict, (iii) looking at the strategy *you*, human, use to read the text '`s : s s s : s : s`'.

Yes, the bottom line is that, ironically, Yacc and Bison cannot use themselves! (More rigorously, they cannot handle the natural grammar describing their syntax.)

Once you have understood the problem, (i) imagine you were the designer of Yacc and propose the grammar for a better syntax, (ii) stop dreaming, you are not Steven C. Johnson, so you can't change this grammar: instead, imagine how the *scanner* could help the parser to distinguish a `symbol` at the right hand side of a rule from the one at the left hand side.

7.12 Further Reading On Parsing

Here a list of suggested readings.

Bison – The YACC-compatible Parser Generator

Written by Charles Donnelly and Richard Stallman
Published by The Free Software Foundation

Available with the sources for GNU Bison. Definitely one of the most beautiful piece of software documentation.

Lex & Yacc

Written by John R. Levine, Tony Mason and Doug Brown
Published by O'Reilly; ISBN: (**FIXME:** *I have the french one :).*)

As many details on all the Lexes and Yacces as you may wish.

Parsing Techniques – A Practical Guide

Written by Dick Grune and Ceriel J. Jacob
Published by the authors; ISBN: 0-13-651431-6

A remarkable review of all the parsing techniques. Freely available on the web pages of the authors since the book was out of print. (**FIXME:** *url.*).

Modern Compiler Implementation in C, Java, ML

Written by Andrew W. Appel
Published by Cambridge University Press; ISBN: 0-521-58390-X

A nice introduction to the principles of compilers. As a support to theory, the whole book aims at the implementation of a simple language, Tiger, and of some of its extensions (Object Oriented etc.).

8 Writing M4 Scripts

9 Source Code Configuration with Autoconf

9.1 What is Autoconf

The following pun remains, in my opinion, the best introduction to Autoconf:

> A physicist, an engineer, and a computer scientist were discussing the nature of God. "Surely a Physicist," said the physicist, "because early in the Creation, God made Light; and you know, Maxwell's equations, the dual nature of electromagnetic waves, the relativistic consequences. . ." "An Engineer!," said the engineer, "because before making Light, God split the Chaos into Land and Water; it takes a hell of an engineer to handle that big amount of mud, and orderly separation of solids from liquids. . ." The computer scientist shouted: "And the Chaos, where do you think it was coming from, hmm?"

And indeed, if you happen to work on a widespread program you will soon learn that there are thousands of subtle or huge differences between systems, which render compilation of a program an impossible exercise. Virtually all the different systems require that sources of a program be adjusted specifically for them.

An impossible task... Instead of providing sources for all the different systems, maintainers program for an ideal system, named POSIX, and use a tool that will try to pretend that the user's machine does fulfill the specifications of this ideal POSIX machine. This program is Autoconf.

The task of Autoconf is therefore to help the maintainer studying the user's machine, diagnosing its non-POSIX-nesses, and providing workarounds. Obviously, examining the user's machine requires running an auditor. Since this auditor is responsible of finding the weaknesses, it must be absolutely universal. The Bourne shell is the only language in which this auditor, `configure`, can be written. But since the Bourne shell has no support for functions (!), programming complex auditors is near to impossible, just as for programming in assembly language.

Higher level languages and compilers have been invented to run away from assembly language. Autoconf, the language, and `autoconf`, the program, have been invented by David J. MacKenzie to run away from plain Bourne shell.

This chapter is a gentle introduction to Autoconf, the tool of reference to ensure software portability. Nevertheless, little emphasize will be put over portability, because:

- solving portability issues is a very tricky activity, which, unfortunately, not only requires cleverness and practice, but also expertise and knowledge. In other words, being smart or even brilliant will never be enough to avoid portability pitfalls: there is a lot to learn by heart.
- fortunately, the GNU/Linux system is POSIX-compliant and essentially bug free. If you aim at publishing software for GNU/Linux, then you really don't need to pay attention to portability issues: everything works fine!

Then, why do you still need to know about Autoconf? First because it is the standard interface to the GNU Build System (which also comprises Automake and Libtool): users are used to run `configure`, to pass '`--prefix=$HOME`' etc. instead of having to edit files by hand. Secondly because POSIX specifies only the core of the system, but if your package requires some specific program or library (say, the GNU Multiple Precision library), then you still have to check for it.

9.2 Simple Uses of Autoconf

Autoconf is merely an M4 library, a set of macros specialized in testing for libraries, programs and so on. Since pure M4 is a bit too low level and since Autoconf aims at producing Bourne shell scripts, it is on top of M4sh, itself atop M4sugar ((**FIXME:** *Ref.*)). As a consequence it has the same syntax: '`#`' introduces comments, the square brackets quote etc. Similarly, you can use '`autom4te --language=autoconf`' to expand Autoconf source files, but for historical reasons and because of the momentum of tradition, everybody runs `autoconf`.

Any Autoconf script must start with `AC_INIT`:

AC_INIT (*package, version,* [*bug-report-address*]) [Macro]
Process any command-line arguments and perform various initializations and verifications. Set the name of the *package* and its *version*. The optional argument *bug-report-address* should be the email to which users should send bug reports.

The simplest Autoconf script is therefore:

```
$ cat auditor.ac
AC_INIT(Auditor, 1.0)
$ autom4te -l autoconf -o auditor auditor.ac
$ ./auditor
$ ./auditor --version
Auditor configure 1.0
generated by GNU Autoconf 2.52g

Copyright 1992, 1993, 1994, 1995, 1996, 1998, 1999, 2000, 2001
Free Software Foundation, Inc.
This configure script is free software; the Free Software Foundation
gives unlimited permission to copy, distribute and modify it.
```

Most packages contain C sources, and therefore need a compiler; `AC_PROG_CC` looks for it:

```
$ cat auditor.ac
AC_INIT(Auditor, 1.1)
AC_PROG_CC
$ autom4te -l autoconf -o auditor auditor.ac
$ ./auditor
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for executable suffix...
checking for object suffix... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
```

It does work, but how can you exploit the results of this system audit? In Autoconf, this is always performed by the creation of new files out of templates. For instance, once your system fully examined, all the '`Makefile.in`' templates will be instantiated for your system into the '`Makefile`'. This is called *configuring* the package, and it is a mandatory tradition to name your configuring script `configure`, created by `autoconf` from '`configure.ac`'.

Transforming a simple auditor into a configuring script requires two additional macro invocations:

AC_CONFIG_FILES (*file...*, [*command*]) [Macro]

Declare '*file*' must be created by copying an input file (by default '*file*.in'), substituting the output variable values. *file* is then named an *output file*, or *configuration file*. If FOO is an output variable which value is 'Bar', then all the occurrences of '@FOO@' in '*file*.in' will be replaced with 'Bar' in *file*.

If given, run the shell *command* once the *file* is created.

AC_OUTPUT [Macro]

Perform all the outputs (create the output files, output headers, etc.).

For instance:

```
$ cat which-cc.in
#! @SHELL@
echo "cc is @CC@"
$ cat configure.ac
AC_INIT(Sample, 1.0)
AC_PROG_CC
AC_CONFIG_FILE([which-cc], [chmod +x which-cc])
AC_OUTPUT
$ autoconf
$ ./configure
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for executable suffix...
checking for object suffix... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
configure: creating ./config.status
config.status: creating which-cc
$ ./which-cc
cc is gcc
$ cat which-cc
#! /bin/sh
echo "cc is gcc"
```

You may wonder what this config.status file is... As a matter of fact, configure is really an auditor, an inspector: it does not perform any configuration action itself, it creates an instantiator, config.status, and runs it.

Output files are not the only form of output: there is also a special form dedicated to providing the results of the audit to a C program. This information is a simple list of CPP symbols defined with #define, grouped together in a *output header*.

AC_CONFIG_HEADERS (*file...*, [*command*]) [Macro]

Declare the output header '*file*' must be created by copying its template (by default '*file*.in'), defining the CPP output symbols.

The traditional name for *file* is 'config.h'. Therefore its default template is 'config.h.in', which causes problems on platforms so severely broken that they cannot handle two periods in a file name. To help people using operating systems

of the previous Millennium, many maintainers use ‘config.hin’ or ‘config-h.in’ as a template. To this end, invoke ‘AC_CONFIG_HEADERS(config.h:config.hin)’ or ‘AC_CONFIG_HEADERS(config.h:config-h.in)’.

If given, run the shell *command* once the *file* is created.

Typical packages have a single place where their name and version are defined: ‘configure.ac’. Nevertheless, to provide an accurate answer to ‘--help’, the main program must know the current name and version. This is performed via the configuration header:

```
$ cat configure.ac
AC_INIT(Audit, 1.1, audit-bugs@audit.org)
AC_CONFIG_HEADERS(config.h)
AC_OUTPUT
$ cat config.h.in
/* Define to the address where bug reports for this package
   should be sent. */
#undef PACKAGE_BUGREPORT

/* Define to the full name of this package. */
#undef PACKAGE_NAME

/* Define to the full name and version of this package. */
#undef PACKAGE_STRING

/* Define to the one symbol short name of this package. */
#undef PACKAGE_TARNAME

/* Define to the version of this package. */
#undef PACKAGE_VERSION
$ autoconf
$ ./configure
$ cat config.h
/* config.h.  Generated by configure.  */
/* Define to the address where bug reports for this package
   should be sent. */
#define PACKAGE_BUGREPORT "audit-bugs@audit.org"

/* Define to the full name of this package. */
#define PACKAGE_NAME "Audit"

/* Define to the full name and version of this package. */
#define PACKAGE_STRING "Audit 1.1"

/* Define to the one symbol short name of this package. */
#define PACKAGE_TARNAME "audit"

/* Define to the version of this package. */
#define PACKAGE_VERSION "1.1"
```

Fortunately, the content of the configuration header template, ‘config.h.in’, can easily be inferred from ‘configure.ac’: look for all the possibly defined CPP output symbols, put an #undef before them, paste a piece of standard comment, and spit it. This is just what autoheader does:

```
$ rm config.h.in
$ autoheader
autoheader: ‘config.h.in’ is created
```

```
$ cat config.h.in
/* config.h.in.  Generated from configure.ac by autoheader.  */

/* Define to the address where bug reports for this package
   should be sent. */
#undef PACKAGE_BUGREPORT

/* Define to the full name of this package. */
#undef PACKAGE_NAME

/* Define to the full name and version of this package. */
#undef PACKAGE_STRING

/* Define to the one symbol short name of this package. */
#undef PACKAGE_TARNAME

/* Define to the version of this package. */
#undef PACKAGE_VERSION
```

"Simple Uses of Autoconf"... Yet there are two programs to run! Does the order matter? (No, it doesn't.) When do I have to run them? (Each time '`configure.ac`' or one of its almost invisible dependencies changes.) Will there be other programs to run like this? (Sure, `automake`, `libtoolize`, `gettextize`, `aclocal`...) Will the order matter? (Yes, it becomes essential.)

"Simple Uses of Autoconf" ought to be named "Simple Uses of `autoreconf`": the latter is a program that knows each of these programs, when they are needed, when they ought to be run etc. I encourage you to forget about `autoconf`, `automake` and so on: just run `autoreconf`:

```
$ rm configure config.h config.h.in
$ cat configure.ac
AC_INIT(Audit, 1.1, audit-bugs@audit.org)
AC_CONFIG_HEADERS(config.h)
AC_OUTPUT
$ autoreconf --verbose
autoreconf: working in `.'
autoreconf: running: aclocal  --output=aclocal.m4t
autoreconf: configure.ac: not using Gettext
autoreconf: configure.ac: not using Libtool
autoreconf: configure.ac: not using Automake
autoreconf: running: autoconf
autoreconf: running: autoheader
autoheader: `config.h.in' is created
$ ./configure
configure: creating ./config.status
config.status: creating config.h
```

9.3 Anatomy of GNU M4ś '`configure.ac`'

The Autoconf world, or better yet, the GNU Build System world is immense. Reading the documentation of these tools (Autoconf, Automake, Libtool) and exercising them on a genuine package is the only means to get used to them. Nevertheless, understanding an '`configure.ac`' is a first required step. This section is devoted to an explanation of a very representative '`configure.ac`': M4's.

As for any non trivially short file, the very first section of M4's 'configure.ac' contains its copyright and license, in comments.

```
# Configure template for GNU m4.  -*-Autoconf-*-
# Copyright 1991-1994, 2000, 2001  Free Software Foundation, Inc.
#
# This program is free software; you can redistribute it and/or modify
...
```

Example : *GNU M4's 'configure.ac' – (i) License*

Then, it requires at least Autoconf 2.53, because it relies on Autoconf and Autotest features available only since then. While this macro was first meant to have a nice error message if your Autoconf was too old, this clause takes an increasing importance these days. Indeed, on systems such as Debian GNU/Linux which ship several concurrent versions of Autoconf, it is used to decide which should be run.

Extreme caution is taken in Autoconf to catch macros which are not expanded. Typically, finding 'm4_define' in the output is highly suspicious. As a matter of fact, any token starting with 'm4_' or 'AC_' is suspected of being an M4 or Autoconf macro that was not expanded, or simply nonexistent. In the present case, since we use variables which names start with 'm4_cv_', we inform the system they are valid. Similarly, to make sure no macro named jm_* (e.g., jm_PREREQ_ERROR) is left unexpanded, we forbid this pattern, and immediately make an exception for variables named 'jm_cv_*'.

```
## ------------------------- ##
## We need a modern Autotest. ##
## ------------------------- ##
AC_PREREQ([2.53])
m4_pattern_allow([^m4_cv_])
# We use some of Jim's macros.
m4_pattern_forbid([^jm_])
m4_pattern_allow([^jm_cv_])
```

Example : *GNU M4's 'configure.ac' – (ii) Requirements*

Then, Autoconf is initialized and the package identified. The invocation to AC_CONFIG_SRCDIR stands for "when running configure, make sure that we can properly find the source hierarchy by checking for the presence of 'src/m4.h'".

The GNU Build System relies on a myriad of little tools to factor workarounds portability issue. For instance, mkinstalldirs is nothing but a portable 'mkdir -p', building a directory and possibly its parents. Such auxiliary quickly populate the top level of a package: 'AC_CONFIG_AUX_DIR(config)' requires to store them in the directory 'config'. The name of this directory is available in configure as $ac_aux_dir.

GNU M4 uses Automake to handle the Makefiles, see Chapter 10 [Automake], page 203. Automake too has to face portability issues: invoking AM_INIT_AUTOMAKE lets it perform the checks it needs. As using configuration headers implies some additional work in the Makefiles, Automake needs that you use AM_CONFIG_HEADERS instead of AC_CONFIG_HEADERS, but it is really the same.

See Section 12.6.2 [Using Autotest with the GNU Build System], page 240, for an explanation of the last two invocations, related to the test suite.

```
## ---------------------------------- ##
## GNU Build System initialization.  ##
## ---------------------------------- ##
# Autoconf.
AC_INIT([GNU m4], [1.4q], [bug-m4@gnu.org])
AC_CONFIG_SRCDIR([src/m4.h])
AC_CONFIG_AUX_DIR(config)

# Automake.
AM_INIT_AUTOMAKE
AM_CONFIG_HEADER(config.h:config-h.in)

# Autotest.
AC_CONFIG_TESTDIR(tests)
AC_CONFIG_FILES([tests/m4], [chmod +x tests/m4])
```

Example : *GNU M4's 'configure.ac' – (iii) Initialization of the GNU Build System*

Because people scattered through out the planet who collaboratively on GNU M4, it is under the control of CVS, see Chapter 13 [Source Code Management], page 243. Since anyone can fetch a snapshot of M4 at any moment, the concept of version number is insufficient. The following section makes sure that non released versions or betas of GNU M4 (which, by convention, end with an "odd letter") have an additional version information: the references of the latest update of 'ChangeLog'.

```
## ---------------------------------------- ##
## Display a configure time version banner. ##
## ---------------------------------------- ##
TIMESTAMP=
case AC_PACKAGE_VERSION in
  *[[acegikmoqsuwy]])
    TIMESTAMP=`$CONFIG_SHELL $ac_aux_dir/mkstamp < $srcdir/ChangeLog`
    AS_BOX([Configuring AC_PACKAGE_TARNAME AC_PACKAGE_VERSION$TIMESTAMP])
    echo
    ;;
esac
AC_DEFINE_UNQUOTED([TIMESTAMP], ["$TIMESTAMP"],
    [Defined to a CVS timestamp for alpha releases of M4])
```

Example : *GNU M4's 'configure.ac' – (iv) Fine version information*

Note that this timestamp is provided to the C code via...

AC_DEFINE (*variable*, [*value* = '1'], [*description*]) [Macro]
AC_DEFINE_UNQUOTED (*variable*, [*value* = '1'], [*description*]) [Macro]
Output '`#define variable value`' in the configuration headers. In the second form, regular shell expansion (back quotes, variables etc.) is performed on *value*.

As a result, `m4` can provide detailed version information:

```
$ m4 --version | sed 1q
GNU m4 1.4q (1.71 Sat, 20 Oct 2001 09:31:12 +0200)
```

Automake provides a set of locations where components of a package are to be installed, e.g., `bindir`, `includedir` etc. We want M4's modules to be installed in an '`m4`' directory in the module directory, `libexecdir`. To this end we define in all the Makefiles a new variable, `pkglibexecdir`, thanks to `AC_SUBST`.

AC_SUBST (*variable*, [*value*]) [Macro]

Substitute '`@variable@`' with its value as a shell variable in the output files. The second argument is a convenient shorthand for:

```
variable=value
AC_SUBST(variable)
```

Then we let the user chose the default modules using an additional `configure` option: '`--with-modules=list-of-modules`'.

```
## ------------------------- ##
## M4 specific configuration. ##
## ------------------------- ##
AC_SUBST([pkglibexecdir], ['${libexecdir}'/$PACKAGE])
AC_SUBST([ac_aux_dir])

AC_MSG_CHECKING(for modules to preload)
  m4_pattern_allow([^m4_default_preload$])
  m4_default_preload="m4 traditional gnu"
  DLPREOPEN=

  AC_ARG_WITH([modules],
    [AC_HELP_STRING([--with-modules=MODULES],
                    [preload MODULES [$m4_default_preload]])],
    [use_modules="$withval"],
    [use_modules="$m4_default_preload"])

  DLPREOPEN="-dlpreopen force"
  if test -z "$use_modules"; then
    use_modules=none
  else
    if test "$use_modules" != yes; then
      for module in $use_modules; do
        DLPREOPEN="$DLPREOPEN -dlpreopen ../modules/$module.la"
      done
    fi
  fi
AC_MSG_RESULT($use_modules)
AC_SUBST(DLPREOPEN)
```

There are several new macros used here. The pair `AC_MSG_CHECKING`/`AC_MSG_RESULT` is responsible of the messages display at `configure` runtime:

```
checking for modules to preload... m4 traditional gnu

## ----------------- ##
## C compiler checks. ##
## ----------------- ##
AC_PROG_CC
AC_ISC_POSIX
```

```
AM_PROG_CC_STDC
AC_PROG_CPP
AC_PROG_CC_C_O
M4_AC_CHECK_DEBUGGING

# Use gcc's -pipe option if available: for faster compilation.
case "$CFLAGS" in
  *-pipe* ) ;;
        * ) AC_LIBTOOL_COMPILER_OPTION([if $compiler supports -pipe],
                [m4_cv_prog_compiler_pipe],
                [-pipe -c conftest.$ac_ext], [],
                [CFLAGS="$CFLAGS -pipe"])
              ;;
esac
## ---------------------- ##
## Libtool initialisation. ##
## ---------------------- ##
AM_ENABLE_SHARED
AC_LIBTOOL_DLOPEN
AC_LIBTOOL_WIN32_DLL
AM_PROG_LIBTOOL
AC_LIB_LTDL

AC_SUBST([LTDLINCL], ["${LTDLINCL-INCLTDL}"])
## --------------- ##
## Gettext support. ##
## --------------- ##
ALL_LINGUAS="cs de el fr it ja nl pl ru sv"
AM_GNU_GETTEXT
AC_CONFIG_FILES(po/Makefile.in intl/Makefile)

if test "$USE_INCLUDED_LIBINTL" = yes; then
  AC_SUBST([INTLINCL], ['-I$(top_srcdir)/intl'])
fi
## ----------------------- ##
## Other external programs. ##
## ----------------------- ##
AC_PROG_INSTALL
AC_PROG_MAKE_SET
AC_PATH_PROG(PERL,perl)
AC_PROG_AWK
## ------------------------ ##
## C headers required by M4. ##
## ------------------------ ##
AC_CHECK_HEADERS(limits.h locale.h memory.h string.h unistd.h errno.h)
AC_HEADER_STDC
## -------------------------- ##
## C compiler characteristics. ##
## -------------------------- ##
AM_C_PROTOTYPES
AC_C_CONST
```

```
AC_TYPE_SIZE_T
AC_CHECK_SIZEOF([long long int])
## -------------------------------- ##
## Library functions required by M4. ##
## -------------------------------- ##
AC_CHECK_FUNCS(bzero calloc strerror tmpfile)
AC_REPLACE_FUNCS(mkstemp strtol xmalloc xstrdup)
if test $ac_cv_func_mkstemp != yes; then
  AC_LIBOBJ(tempname)
fi
AC_FUNC_ALLOCA
AC_FUNC_VPRINTF

AM_WITH_DMALLOC

jm_PREREQ_ERROR

M4_AC_FUNC_OBSTACK
M4_AC_SYS_STACKOVF

M4OBJS=
m4_pattern_allow([^m4_getopt_h$])
m4_getopt_h=src/getopt.h
rm -f $m4_getopt_h
AC_CHECK_FUNC([getopt_long], [],
    [M4OBJS="getopt1.$ac_objext getopt.$ac_objext"
    AC_CONFIG_LINKS([$m4_getopt_h:src/gnu-getopt.h])])
AC_SUBST([M4OBJS])


# This is for the modules
AC_STRUCT_TM
AC_FUNC_STRFTIME
AC_CHECK_FUNCS(getcwd gethostname mktime uname)
AC_CHECK_FUNCS(setenv unsetenv putenv clearenv)

AC_LIB_GMP
AM_CONDITIONAL([USE_GMP], [test "x$USE_GMP" = xyes])
## ---------------------------------- ##
## Make sure LTLIBOBJS is up to date. ##
## ---------------------------------- ##
Xsed="sed -e s/^X//"
LTLIBOBJS=‘echo X"$LIBOBJS" | \
    $Xsed -e ’s,\.[[^.]]* ,.lo ,g;s,\.[[^.]]*$,.lo,’‘
AC_SUBST([LTLIBOBJS])
## -------- ##
## Outputs. ##
## -------- ##
AC_CONFIG_FILES(Makefile config/Makefile doc/Makefile m4/Makefile
                m4/system.h:m4/system-h.in src/Makefile modules/Makefile
tests/Makefile examples/Makefile)
```

```
AC_OUTPUT
```

9.4 Understanding Autoconf

Teaching to the reader the long road to Autoconf guruness is way beyond the scope of this book. Exploiting the full power of Autoconf is, unfortunately, reserved to the few people who are ready to spend hours tracking portability issues, as mastering Autoconf stands for mastering portability issues.

Nevertheless, we feel you ought to be revealed a few secrets about Autoconf.

9.4.1 Keep It Stupid Simple

It is unfortunate that the most important rule is still a secret today:

Never try to be smart with Autoconf

Many people write '`configure.ac`'s that are rejected by different versions of Autoconf, or will be rejected in the future. The Autoconf maintainers often receive complaints about such problems, but they are really introduced by the users themselves.

The first most common problem is relying on undocumented features. You should **never** do that. If it is undocumented, it is private, and likely to be changed in the future. The most frequent reason to rely on undocumented feature is to save some typing: you have to address a task, and notice some internal macro performs a job close to your needs. Don't use it: either you copy and adjust it to your very needs —under a different name of course—, or you ask to the maintainers to make a public version of this macro.

The worst case is with people who want to rely on very low level details, or even in some cases, *change* some low level macros! This is doomed to failure. There are several reasons making maintainers try this perverse game:

bypass the official interface
: Autoconf follows the GNU Coding Standards, which some people sometimes find painful —for instance because they want options that do not fall into the GNU standard set of options for `configure`. You should rely stick to these standards, experience has proved that they cover all the needs, possibly in an admittedly convoluted way. And if they don't, then ask for changes in the GNU Coding Standards: Autoconf will follow.

adjust existing macros to different needs
: Many people want to hook their code onto Autoconf macros. For instance, "when `AC_PROG_CC` is called I want `MY_PROG_CC_HOOK` to be invoked". You cannot imagine the complex tissue of interdependencies that already exists in Autoconf! Checking for a compiler for instance, requires relying on many different preliminary initializations and checks. The following figures should give you an idea of this amount of work: `AC_PROG_CC` alone produces more than 20Kb of code, almost 900 lines of shell script! And this code is not contiguous: it goes into three different sections of `configure`.

  Don't try to hook your macros: just invoke them. Sure, your change is not longer "invisible", the user must call it explicitly, but at least it will be robust.

If you see no option to address your need, ask the Autoconf maintainers: either the know the right way to do it, or they will provide you with a new macro in the official Autoconf.

The second most common problem is trying to optimize `configure`. For instance they skip long series of tests needed only for some features the user did not chose. This exposes you to extremely nasty, stealthy, vicious bugs. Unless you know exactly what you do (I am here referring to people who have an exact knowledge of Autoconf), *never perform tests conditionally: depend conditionally on their output!*

Here is a simple example of such an broken "optimized" '`configure.ac`':

```
AC_INIT

AC_ARG_WITH([fprintf])
if test "x$with_fprintf" = xyes; then
  AC_CHECK_FUNCS(fprintf)
fi

AC_ARG_WITH([sprintf])
if test "x$with_sprintf" = xyes; then
  AC_CHECK_FUNCS(sprintf)
fi
```

The following runs clearly demonstrate the point of this optimization:

```
$ ./configure
```

as nothing was needed, nothing was checked for. If using `fprintf` is requested, then of course, we need a C compiler to check for its existence, and then check for it:

```
$ ./configure --with-fprintf
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for executable suffix...
checking for object suffix... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for fprintf... yes
```

Similarly if both `fprintf` and `sprintf` are needed:

```
$ ./configure --with-fprintf --with-sprintf
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for executable suffix...
checking for object suffix... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for fprintf... yes
checking for sprintf... yes
```

As expected, `fprintf` and `sprintf` are both available on my GNU/Linux system.

Now, sit back, and look at this:

```
$ ./configure --with-sprintf
checking for sprintf... no
```

although `sprintf` *is* present!

What happened is that Autoconf knows that checking for a function requires a compiler for the current language (here, C), so it actually expands something similar to the following '`configure.ac`':

```
AC_INIT

AC_ARG_WITH([fprintf])
if test "x$with_fprintf" = xyes; then
  AC_PROG_CC
  AC_CHECK_FUNCS(fprintf)
fi

AC_ARG_WITH([sprintf])
if test "x$with_sprintf" = xyes; then
  AC_CHECK_FUNCS(sprintf)
fi
```

As a consequence, if `fprintf` is not requested, `configure` will not look for a C compiler, and all the following tests are broken. Never run tests conditionally: depend conditionally on the *results* of the tests:

```
AC_INIT

AC_ARG_WITH([fprintf])
AC_CHECK_FUNCS(fprintf)
if test "x$with_fprintf" = xyes; then
  # Depend on the presence/absence of fprintf here.
fi

AC_ARG_WITH([sprintf])
AC_CHECK_FUNCS(sprintf)
if test "x$with_sprintf" = xyes; then
  # Depend on the presence/absence of sprintf here.
fi
```

10 Managing Compilation with Automake

11 Building Libraries with Libtool

12 Software Testing with Autotest

This chapter is devoted to test suites, i.e., programs which are meant to exercise other programs in order to perform sanity checks, to prevent old bugs from creeping back in etc.

While programmers understand their jobs involve more than programming, most still do not pay the attention to the test suite that it deserves; the Section 12.1 [Why write tests?], page 207 advocates for test suites under the form of three buggytales. Then, in Section 12.2 [Designing a Test Suite], page 211, some of the generic rules to obey while implementing a test suite are presented. See Section 12.4 [Running an Autotest Test Suite], page 217, for a presentation of Autotest, the Autoconf component dedicated to portable test suite generation. For a more "hands on" presentation of Autotest, see Section 12.5 [Stand-alone Test Suite], page 219, which demonstrates Autotest features, step by step, applied to M4. Finally, in Section 12.6 [Autotesting GNU M4], page 235, the actual GNU M4 test suite is pictured, exhibiting all the characteristics of real world Autotest uses.

12.1 Why write tests?

Everybody will agree with the usefulness of writing tests, but in practice it takes some time before novice programmers pay attention to them. See Section 12.1.1 [Joe Package Version 0.1], page 207, for a story many of us lived as the main character. That's only the beginning of the story, intent to design a test suite is not enough, and even the most experienced programmer may be caught by bugs deserving an appearance in a Monty Python movie. See Section 12.1.2 [Fortran and Satellites], page 209, for a demonstration of the importance of realism in tests. Unfortunately there is no silver bullet against "*errare human est*", and even using the strictest development procedures in the world, there is no protection against bugs *within the testing framework*, see Section 12.1.3 [Ariane 501], page 210, for a $500 000 000 fireworks story.

12.1.1 Joe Package Version 0.1

Joe is a novice maintainer. He installed his first GNU/Linux system last year, played a bit with it, experienced the programming environment, and finally discovered that his system was lacking a little something which would make his life much easier.

He started writing a small shell script to fulfill some of his needs. Along the months his shell script grew big, for it included all the features he needed, plus some bells and whistles he is proud of. But it became slower and slower, almost unusable.

Based on his experience, he redesigned his project, and implemented it in C. Some of his friends, discovering his program, realized they really needed it on their GNU/Linux machine, and even wanted more features. Soon he was to package his project, which met an immediate and unexpected success: the statistics of his web page revealed an average of one hundred downloads per day.

Soon the trouble started.

Of course there were people who could not compile his project, because of missing or broken functions in their C library. Those are the easy problems, which he quickly solved thanks to Autoconf.

Later someone reported a segmentation fault when using his package. It took Joe a couple of message exchanges to get some fundamental information, such as the version of the package and the command line which triggered the segmentation violation. Finally, after having delivered a quick lecture on `gdb`, he managed to get a stack trace from the user, and the value of some of the variables. The library files loading failed because their directory was `NULL`. After some more

messages, more and more delayed since the user grew tired of running a debugger, Joe finally had an idea, and asked the user to send his configuration file.

The user had edited it, and the '`LibraryDirectory:`' line was lacking...

Joe equipped his program with additional sanity checks, to make sure such variables are set, and released another version of his package.

Someone else reported some unexpected behavior at runtime; the program did not crash, but systematically refused to work properly, complaining about the absence of files although they *were* present! Nothing made sense, neither to Joe nor to his user.

Yet another series of messages, the first of which asking the version of his package, the command line the user typed and... the configuration files. A close examination of all this data gave no hint of what might have gone wrong. It took Joe two weeks and many messages, which included his now usual `gdb` lecture notes, to finally discover his user was running some system with a different encoding of end-of-line! Raging that the user never reported the system he runs, but now understanding the problem, he quickly solved it, and made another release.

Later a system administrator sent him angry messages: under the pressure of his users who are fond of Joe's package, he installed the newly published "bug fix" release; they all started complaining nothing worked at all, the package was completely unusable. Hurt in his self-esteem, Joe first answered that his package was delivered with NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE, but after a few messages asked the usual questions: the version of the package, the command line, the configuration files, and the architecture. This time it went fast, since the system administrator was experienced and quickly localized the bug in some experimental code. This code was under development and compiled only when '`--enable-experimental-feature`', a crucial information lacking from the initial bug report. Reinstalling the package without the option was enough.

Joe learned his lesson well, and added a section "Reporting Bugs" to the documentation. He even added some tracing options to his package which will definitely help him locating problems, that later appeared to be often due to unexpected behavior from the users; he is now happily using an expression he grew fond of: UBD, User Brain Damage[1]. But he was to realize that few users actually read that section, and he regularly has to refer them to it.

Yet another release.

Yet another problem: the very same administrator, now a friend (which is always what happens after long sessions of virile bug hunting expeditions), reports an endianness problem was introduced with the tracing options. He even provides a patch!

Fix. Release.

It's been three months since the initial release. In the meanwhile, he had many new and exciting ideas of new cool features, but he had no time to implement any of them, only the preliminary version of one of them that caused him some troubles. Looking back to these three months, he realized that he spent all his spare time to fixing bugs, to answering bug reports, most of the time just asking the user some crucial information, sometimes teaching them how to run a debugger to trace some execution. It was not rare that the user stopped providing feedback before the origin of the bug was actually found, resulting in a pure loss of time for

[1] According the Jargon dictionary:

UBD /U-B-D/ /n./ [abbreviation for 'User Brain Damage']. An abbreviation used to close out trouble reports obviously due to utter cluelessness on the user's part. Compare "pilot error"; oppose PBD.

pilot error /n./ [Sun: from aviation]. A user's misconfiguration or misuse of a piece of software, producing apparently buglike results (compare UBD). "Joe Luser reported a bug in sendmail that causes it to generate bogus headers." "That's not a bug, that's pilot error. His '`sendmail.cf`' is hosed."

PBD /P-B-D/ /n./ [abbrev. of 'Programmer Brain Damage']. Applied to bug reports revealing places where the program was obviously broken by an incompetent or short-sighted programmer. Compare UBD.

both, and the terrible frustration of knowing there is a bug somewhere, and not being able to know where.

What he was lacking is, obviously, a good test suite.

Tracking bugs can swallow all your time. You cannot foresee all the variations of the systems on which your package will be installed. You cannot imagine how heavily a daily use by thousands of users can exercise your code in unexpected ways. The main purpose of a test suite is to save both the users' and your time. It must be able to denounce problems *before* the package is installed to avoid breaking a working (previous) installation. It should be usable without any kind of expertise. It should help the user sending the maintainers all the information they might need: version of the package, configuration options, as much data as possible on the architecture and the environment, traces etc.

12.1.2 Fortran, Antennae and Satellites

(**FIXME:** *Can't find the reference to this adventure..*)

In Fortran an identifier doesn't need to be declared:

- if it starts with an '`i`', '`j`', '`k`', '`l`', '`m`', or '`n`', it's an integer variable;
- if it starts with another letter, it's a float variable;
- white spaces are insignificant (they are not separators).

A satellite driving software was written in Fortran; one of its tasks was extending the antenna once on orbit. An excerpt of the program that was to be written is presented below, together with it C equivalent for our non Fortran fluent readers:

```
                              int I;
                              ...
  DO 1 I = 1, 5               for (I = 1; I <= 5; ++I)
                              {
C Extend the antenna.           /* Extend the antenna. */
  ...                           ...
C If antenna is extended        if (antenna_is_extended)
C Then Go to 2                    goto 2
  ...                           ...
1 CONTINUE                    }
2 ...                         2:...
```

Example 12.1: *Extending a Satellite Antenna in Fortran*

This code intends to try five times to open the antenna before giving up. When tested on the ground, the antenna always opened at the first try.

Unfortunately the actual program had a dot instead of a comma in the loop statement, and *because of a single character typo* the meaning is completely different:

```
                                  float DO1I;
                                  ...
      DO 1 I = 1. 5               DO1I = 1.5;

    C Extend the antenna.         /* Extend the antenna. */
      ...                         ...
    C If antenna is extended      if (antenna_is_extended)
    C Then Go to 2                  goto 2
      ...                         ...
   1 CONTINUE
   2 ...                          2: ...
```

Example 12.2: *Not Extending a Satellite Antenna in Fortran*

The antenna was not extend at the first and only try, the satellite was lost.

12.1.3 Ariane 501

On June 4, 1996, the Ariane 5 maiden flight 501 failed 40s after its takeoff with the explosion of the launcher. The amount lost was $500 000 000, five hundred billion dollars. Uninsured.

(**FIXME:** *Ariane: Heck. ESA seems to have withdrawn the report from the Web, what reference shall I put now? There remains the French version of the CNES..*)

The explosion resulted from a software error in the Inertial Reference System (SRI[2]) software module. Ariane 5 reuses the SRIs and much of the hardware and software from Ariane 4, her elderly sister, and in particular the SRIs modules were kept as is, since they proved to be perfectly reliable over the last ten years.

Nonetheless, 37 seconds after takeoff, the two SRIs software modules detected an overflow: the horizontal bias of the flight, measured on 64 bits, no longer fitted in a 16 bit integer. The exception handling mechanisms were triggered, but since this overflow was not caught, the default exception handler of both modules concluded an impossible situation, diagnosing severe problems: they shut down. This resulted in Ariane veering abruptly, and in the launcher properly committed suicide. Although the trajectory was perfect.

The SRIs failed during the very first flight of Ariane 5 while they worked perfectly for Ariane 4. How come? Because the trajectories of the two generations of Ariane are completely different.

The overflow was not detected during simulations and testing. How come? Because testing has also been performed *using Ariane 4 trajectories.*

The SRIs were still running 37 seconds after takeoff while they are useful only *before.* How come? The two SRIs are normally shut down 9 seconds before takeoff, but in the event that the count down is held just before takeoff, to avoid the additional delays needed to reset the SRIs, which would delay the whole launch for hours, they are kept alive until 50 seconds after takeoff.

They trusted a single pair of modules while everybody knows that in planes critical systems are tripled and election is performed on the results. How come? Because they *did* double the SRIs modules, but it was a mere duplication, and both twins correctly detected the failure at the same time.

The recommendations given by the committee of experts which analyzed the failure include:

[2] SRI, standing for the French "Système de Référence Inertiel", is the term used by the European Space Agency in its English reports. Maybe to avoid evoking bad memories for American people...

– *improvement of the representativeness (vis-a-vis the launcher) of the qualification testing environment;*
– *introduction of overlaps and deliberate redundancy between successive tests (i) at equipment level, (ii) at stage level, (iii) at system level;*
– *switch-off or inhibition of the* SRI*s alter lift off;*
– *testing to check the coverage of the* SRI *flight domain;*
– *general improvement of representativeness through systematic use of real equipment and components wherever possible;*
– *simulation of real trajectories on* SRI *electronics.*

12.2 Designing a Test Suite

Any experienced programmer knows her job stretches much further than merely typing instructions in a programming language. Just as programmers need pressure and experience to appreciate the importance of documenting their code, they need to be educated about the importance of testing. Writing tests is usually considered as a waste of time, and sometimes even a loss of "productivity".

Unfortunately, most test suites are often simply shell scripts written by hand, which is indeed not very "productive". There exist a few testing frameworks that ease the maintenance, in particular Autotest. They make it possible to *design* a test suite.

12.2.1 Specify the Testing Goals

A test suite is a project in and of itself.

As much as possible, you should define the objectives to be reached by the test suite, and you even may isolate different aspects that should be covered by different test suites, or even different test tools.

Knowing who will run the test suite is crucial: a maintainer runs additional tests and only needs raw information such as a core dump, while when run by a user it should ease the writing of a precise bug report.

Keep in mind that message exchanges are extremely costly in the bug tracking process: the user is likely to be lax about answering simple questions. Without any answer, you will know a bug is sneaking, imperceptible to you, in the program.

You might come up with different test suites, some of them exercising the code with time consuming internal checks activated, or simply changing something in the environment. For instance you might want to run your programs with special tools which considerably improve the severity of the testing conditions that the users do not necessarily have: debugging memory management libraries, bound checking compilers, etc.

Running all these tests can be extremely time consuming[3], and can discourage users from running them. While this should never be a reason not to test a feature, you should nonetheless keep this factor in mind, and maybe design a consistency test suite which is to be run by maintainers only, and a "public" test suite, which would be run by virtually all the users.

[3] The GCC test suite takes hours to run on most architectures. While performed in a quarter of an hour on my machine, the Autoconf test suite has been reported to take up to 9 hours by some users. Using Dmalloc at its maximum checking possibility slows down a program by several orders of magnitude, making it hardly usable.

12.2.2 Develop the Interface

A test suite is a tool.

Because programs are sensitive to the environment[4], playing with a failure is often needed, either to get more details using some specific feature, or after a variable was changed, and simply after the program is —hopefully— fixed. All these scenarios may happen on the user's side, therefore you should make it easy for them: have a clean interface, and some user documentation. Bug reporters are well-meaning, but have often little time and varying skills: let your interface be simple enough for you to ask "please run this simple command". Conversely, some users are ready to spend some time tracking the origin of a failure: everything must be done to play easily with variation of the test scenario.

Because unfortunately not all the failures are detected by the test suite, you might also be interested in equipping the programs with verbosity/tracing options. Many programs provide a '`--verbose`' option which produces information about critical internal variables, the various actions that are performed etc. It often helps locating *when* the bug was triggered.

No one would ever work with a Boolean compiler: while '`compilation succeeded`' is all we need, '`compilation failed`' is definitely not enough information for a programmer to track an error. Test suites are very similar: we all expect '`test suite succeeded`', but '`test suite failed`', or even '`test suite failed: test 51 failed`', are not enough. Your testing scheme should provide accurate information on the tests that were performed, what was expected, and what was obtained.

If the test suite may be run by users, then you should pay even more attention to providing as much information as needed to understand a failure *at distance*. For instance, have the test suite wrap all the pertinent information in a log file, and ask for this file.

12.2.3 Look for Realism

A test suite is a user.

See Section 12.1.3 [Ariane 501], page 210, for a demonstration of the importance of realism in tests. While testing from the inside (with consistency checks or even tests embedded in the executables themselves) is a precious means to catch failures as early and as precisely as possible, realism should always be on your mind. There are always surprising bugs when putting together features which, individually, work perfectly. Only real uses are likely to reveal them. As a consequence, don't test the package, *use* it!

Bison (see Section 7.3 [What is Bison], page 162) is a generator of C files, and a lot of testing can be performed by simple checks performed on its input: looking for specific lines etc. But this is not a real use, you'll miss a lot of errors that way; the C output has to be exercised itself on inputs that stress the parser.

Bypassing the official interface of the tools is extremely tempting, as it usually makes it faster to write the tests or speeds up the test suite. But sooner or later you might pay for that simplification, either because you missed a bug triggered on actual input, or because some inner detail of the program changed invalidating the test itself. A valid and complete sequence would have kept the test valid since the user interface is often kept backward compatible.

[4] Like for plants and fishes, have your sensitive programs listen to classical music instead of hard core techno. Do as I do: use earphones.

I once released a broken package: some of its files were not installed. The test suite did not (and could not!) diagnose it. Why? Because I wanted the test suite to run in the user's directory, where the user built my package, before it was installed. To make this possible, the test suite set a lot of environment variables, skipped the regular `PATH` use etc. Then of course, it was finding the files! It's almost as if it were directly looking for them in the tarball.

My test suite was not a regular user, and because of this I could not use it on an installed version of my program. Actually, most test suites have exactly this problem: they are not regular users, they are biased users exercising the package *under specific conditions*.

Now my test suites rely on `PATH` exclusively, like regular users. But then, how to test a package before it is installed? First, add small wrappers in your package, typically shell-scripts that set environment variables and then run the real not yet installed binaries. Then run the test suite after having set the `PATH` so that these wrappers are found first. Simple enough!

Be sadistic, be mean! Anything that can strengthen the test suite should be used. If your programs have self-testing features, or simple sanity checks such as a '`--warning`' option, use them. Some users have extremely surprising expectations, or are simply very demanding; they might hit some limitations in your package. Precede such uses and write *torture tests*, stretching the limits as far as possible. Many bugs lie in the angles, at the extremes of the range of validity of your routines: who has never been bitten by a '`<`' where a '`<=`' was needed? The number of stupid bugs, silly "lacking the room for the trailing \0" errors that torture tests catch is impressive[5]. It might even help realize you were about to waste a satellite because of a single character typo, as recounted in Section 12.1.2 [Fortran and Satellites], page 209, something the tests did not reveal for lack of realism.

Users do make mistakes (occasionally). Not only should you exercise the programs at the extreme of their validity domains, but you should also test them on invalid situations. A program which fails to properly reject invalid situations *is* broken. If you check only for valid conditions you might release a program dying when given a nonexistent filename. Sooner or later, you'll have to waste time answering zillions of similar SEGV reports while a simple '`No such file or directory: 10^X^F`', or '`not a number: 10:wq`' would have saved you from this hassle.

12.2.4 Ordering the Tests

A test suite is a tool.

A test suite must be designed to assist you when something goes wrong. If you merely append test cases one after the other, then some day you will receive a huge log in which possibly 90% of the tests failed. Obviously some low level routine is not working properly for this configuration, but which one? With which one of the hundred of failing test should you start? What is their most probable common origin?

Tests, within a test suite, shall be built just as the programs themselves: if the program consists of layers of modules, or simply layers of routines, then exercise the low level layers first. In other words, exercise bottom up. Then, chances are high that addressing massive test suite failures from the first failures to the last will be the shortest path to a properly fixed program.

The patience of the user, and the increasing likelihood of his interrupting the test suite, are also to be taken into account. If you have torture tests (and you should) then putting them last

[5] Early Macintosh users might remember the so-called Monkey test: a simple program was randomly moving the mouse, clicking here and there. I don't remember having to wait for more than a few minutes to have to manually reboot my computer. Pose, the PalmOS emulator provides the same feature under the name of "Gremlins mode".

diminishes their chances of being run, hence your chances to learn that under extreme conditions your package fails.

Unfortunately the two objectives, ordering programmo-morphologically and usero-impatiencely, are often incompatible since torture tests usually involve as many parts of the software as possible, while bottom-up testing emphasizes single component testing.

Autoconf faces this dilemma. Torture tests are critical for Autoconf, since they are meant to guarantee portability of complex requests across all the exotic systems some users have, and across all the creativity of some maintainers. If some sed's limitations are hit by Autoconf, then it must be known it before some fundamental package such as Emacs is found to be impossible to install on some systems. But these torture test failures are extremely hard to analyze...

In the case of Autoconf, we chose to exercise the most fundamental features first, then the torture tests, and finally automatically generated tests, which are representative of the most typical uses. Up to now this order proved to be efficient, as grave failures are detected early, and it only happened a couple of times that the failure of torture tests be understood thanks to tests run afterwards.

12.2.5 Write tests!

A test suite is a sister project.

A test suite is a project, sharing a special relationship with the core project:

- implement tests for new features as you implement them;
- implement tests for new bugs as you deimplement them.

Do you know of any programmer who thinks of a new feature, implements it and immediately releases it? I don't know of any, with one exception: students who sometimes manage to deliver code which doesn't even compile.

All the programmers *exercise* their code *while* they implement it. Very young programmers, first year students (first month students actually) spend hours typing the same set of values to check that their implementation of quicksort work properly. Slightly older programmers (second year students, or students who stayed down) quickly learn to write input files and use shell redirections to save their efforts. But they usually throw away these test cases, and as the project gets bigger, they suddenly observe failures of features that were working properly months ago. Older programmers keep these test cases. Experienced programmers comment, document and write tests while they implement (see Section 12.2 [Designing a Test Suite], page 211, and Literate Programming). Some authors recommend that developers spend 25-50% of their time maintaining tests (**FIXME:** *Should I ref this? http://www.xprogramming.com/testfram.htm.*).

Don't be bitten three times by the same dog! Write *regression tests*.

While most bugs probably do not need to be tracked down by dedicated tests, at least they demonstrate that some high level test is missing, or is not complete. For instance a bug was found in Bison: some C comments were improperly output like '`//* this. */`'. A dedicated test was written. This is overkill. It demonstrated that the high level tests, exercising the full chain down to the generated executable, needed to include C comments. Not only was this overkill, but it is also quite useless: this bug is extremely unlikely to reappear as is, while it is extremely likely that at other places, comments are also incorrectly output. The test suite was adjusted so that the sources be populated with comments at all sorts of different places.

When you spent a significant amount of time tracking the failure of a feature in some primitive problem, immediately write a dedicated test for the latter. Do not underestimate the importance

of sanity checks within the application itself. It doesn't really matter whether the application diagnoses its failure itself or whether the test suite does. What is essential is that the test suite exercises the application in such a way that the possible failure be tickled.

You should always write the test before fixing the actual bug, to be sure that your test is correct. This usually means having two copies of the source tree at hand, one running the test suite to have it fail, and the other to have the same test suite succeed.

If you track down several bugs down to the same origin, write a test especially for it.

Of course in both cases, more primitive tests should be run beforehand.

Test generation, or rather test extraction[6], is a valuable approach because it saves effort, and guarantees some form of up-to-dateness. It amounts to fetching test cases from the documentation (as is done in GNU M4 for instance), or from comments in the code, or from the code itself (as is done by Autoconf).

More generally, look for means to improve the maintainability of your test suites.

12.2.6 Maintain the Test Suite

A test suite is a project in its own right.

. . . and therefore demands to be maintained. If you don't, it will become useless or unmaintainable, just like any other kind of program. Spend some time to:

- generalize specific tests;
- improve the maintainability of the test suite.

Try to generalize your specific tests when you implement or improve tests. For instance, if you are testing a feature which has a fixed set of possible values, test them all. If you exercise the interaction between two such features, do not hesitate the test the Cartesian product of their values, i.e., the set of all the valid *and* invalid couples.

(**FIXME:** *I should first ask Tom if he agrees with the following paragraph..*)

The Automake test suite is a good example of what should not happen. Automake supports some form of conditionals, which is a typically feature with a small set of possible values: true and false. Conditionals can interact with each others, since they can influence the same set of variables and/or targets. Because it turned out to be much more delicate to implement than one may first think, the implementation was often changed. Virtually all the modifications were bug fixes, but they often introduced new ones. Gradually the test suite covered more and more cases of conditional uses, and today they cover almost the full range of possible values, the very Cartesian product aforementioned. But this coverage is performed via several handwritten tests, which are modified copies of the previous tests: merely checking that the coverage is complete is a delicate task because of the lack of homogeneity across these tests.

If the first test author had devoted some more time to his test, not only would the improvement of conditionals would have been sped up, but the testing framework would also have been improved because it would have been developed with generalization in mind. This is to parallel with novice programmers preferring to copy-paste-modify a routine n times for n slightly different tasks as compared to the generalization of existing routines to cover these n cases. Which brings us to our next point...

[6] *Automatic Test Generation* usually refers to the generation of tests from formal specifications of a program, and possibly from the program itself. *Formal specifications* are written in a mathematical language, typically set theory, and describe very precisely the behavior of a program.

Expertise is gained during the test suite life time, and its rethinking is often beneficial. Just as a regular project, common patterns arise, and factoring can be done. Your test framework should support some form of programming so that this very factorization be possible.

Conventional Bourne shell based tests are again an excellent example of what should not be done. Automake, again, suffered from this: because there is function support in Bourne shell, there is a lot of code duplication, which results in sometimes having to repeat the same modifications on many different files (there are more than 300 test files). I personally had to change several times more than a hundred tests to cope with Automake performing some better sanity checks: these tests, which bypassed the official interface, were no longer "correct" Automake users (see Section 12.2.3 [Look for Realism], page 212): *the test suite must be viewed as a user*).

It is common that these factorizations, these new test functions or macros, reveal holes in the testing. Reading seven invocations to a general routine testing a feature makes it easy to find the eighth case was lacking. Seven test cases written differently, at different places in the test suite, make it impossible for the maintainer to complete its coverage.

12.2.7 Other Uses of a Test Suite

In the previous sections, and in particular Section 12.2.3 [Look for Realism], page 212, we emphasized the fact that a test suite is a user. As a result, a test suite is no less than a set of uses of your package, a *corpus* linguists would say.

It can then become a good set of samples on which profiling your package (see Chapter 15 [Profiling and Optimising Your Code], page 263), much more relevant than a few runs by hand.

There exist compilers that optimize a program thanks to profiling information[7]: they first compile the program making some more or less arbitrary choices, and then the program can be recompiled using logs produced by several runs to make better choices. Again, the test suite, and especially the torture tests, provide a good set of uses for profile guided compilation.

12.3 What is Autotest

The previous section highlighted that test suites are actual projects: they have to be maintained, extended etc. To ease their maintenance, there are a few tools, most notably DejaGNU. Unfortunately running a DejaGnu test suite requires DejaGNU, which itself requires TCL! As a consequence, given that few users installed DejaGNU on their machines, the DejaGNU test suites are rarely run, severely reducing part of their interest: exercising a package on a wide variety of platforms.

To make sure their users will always be able to run their test suites, many package maintainers write their test suite as a collection of hand written Bourne Shell scripts. Needless to say that this is long, tedious, unmaintainable etc.

History already faced this situation and provided an answer: people used to write long portable shell scripts to *configure* their package. Autoconf was invented to automatically create these configuring scripts from synthetic descriptions. Similarly, Autotest was invented to automatically create testing scripts. Autoconf is a configuration scripts compiler, Autotest is a testing scripts compiler. Because they share a significant part of code, Autotest is part of the package Autoconf.

[7] There are several optimization kinds which face the *combinatorial explosion*: there are many different possibilities amongst which one or several are better than others. Finding an optimum efficiently is then impossible and approximations are used. Sometimes the concept of "optimum" is bound to the uses (this choice is better for these uses, that other choice is better for those other uses). In either case, tuning the choice thanks to actual uses improves the average efficiency.

An Autotest test suite is a series of test groups. A *test group* is a sequence of interwoven commands that ought to be executed together, usually because one creates data files that a later test in the same group needs to read. For instance, a Bison test group is typically composed of three tests: one which runs `bison`, one which runs the compiler, and one that runs the generated executable. They form a consistent group: it makes no sense to run the last step if the previous steps were not, nor to run the last steps if the previous ones failed.

In the following section, Section 12.4 [Running an Autotest Test Suite], page 217, we present the Autotest test suite from the user's point of view: "I'm installing a new package on my machine: how do I run its test suite?". The rest of the chapter is dedicated to writing and compiling an Autotest test suite.

12.4 Running an Autotest Test Suite

An Autotest test suite is often named `testsuite`, but we advertise for more explicit names, such as `test-m4`, `test-joepackage` etc. This way, it can be installed or sent to other people.

To test a package, such as GNU M4, just run its test suite:

```
$ cd m4-1.5/tests
$ ./testsuite
## ---------------------- ##
## GNU m4 1.5 test suite. ##
## ---------------------- ##

Macro definitions.

  1: macros.at:29      ok
  2: macros.at:71      ok
  3: macros.at:105     ok
...
Options.

 21: options.at:26     ok
 22: options.at:52     ok

Composite macros.
...
Documentation examples.

 43: generated.at:14   ok
...
 78: generated.at:1404 ok
 79: generated.at:1438 ok
## ---------------------------- ##
## All 79 tests were successful. ##
## ---------------------------- ##
```

All the test groups (here, numbered from 1 to 79) are run. If some failed, a log file would have been created, full of details that might help the GNU M4 maintainer to understand the failure. For instance in the following example, I have manually changed the test group 44 for it to fail:

```
$ ./testsuite 44
```

```
## ---------------------- ##
## GNU m4 1.5 test suite. ##
## ---------------------- ##
 44: generated.at:48   FAILED near `generated.at:60'
## ------------------------------------------------ ##
## ERROR: Suite unsuccessful, 1 of 1 tests failed. ##
## ------------------------------------------------ ##
You may investigate any problem if you feel able to do so, in which
case the test suite provides a good starting point.

Now, failed tests will be executed again, verbosely, and logged
in the file testsuite.log.
## ---------------------- ##
## GNU m4 1.5 test suite. ##
## ---------------------- ##
44. generated.at:48: testing Define...
generated.at:60: m4 -b -d input.m4
--- -   Sat Jan 12 17:31:38 2002
+++ at-stdout     Sat Jan 12 17:31:38 2002
@ -1,3 +1,3 @

-Hello world.
+Hello universe.

44. generated.at:48: FAILED near `generated.at:60'
## ------------------------ ##
## testsuite.log is created. ##
## ------------------------ ##

Please send `testsuite.log' to <bug-m4@gnu.org>,
along with all information you think might help.
```

Autotest test suites support the following arguments:

`'--list'`
`'-l'` List all the tests (or only the selection), including their possible keywords.

To change environment variables, set of tests, or verbosity of the test suite:

`'variable=value'`
: Set the environment *variable* to *value*. The variable `AUTOTEST_PATH` specifies the testing path to prepend to `PATH`.

`'number'`
`'number-number'`
`'number-'`
`'-number'` Add the corresponding test groups to the selection.

`'--keywords=keywords'`
`'-k keywords'`
: Add to the selection the test groups which title or keywords (arguments to `AT_SETUP` or `AT_KEYWORDS`, see (**FIXME:** *Test Groups.*)) match *all* the keywords of the comma separated list *keywords*.

  Running '`./testsuite -k autoupdate,FUNC`' will select all the tests tagged with '`autoupdate`' *and* '`FUNC`' (as in '`AC_CHECK_FUNC`', '`AC_FUNC_FNMATCH`' etc.) while

'`./testsuite -k autoupdate -k FUNC`' runs all the tests tagged with '`autoupdate`' *or* '`FUNC`'.

'`--errexit`'
'`-e`' If any test fails, immediately abort testing. It implies '`--debug`'.

'`--verbose`'
'`-v`' Force more verbosity in the detailed output of what is being done. This is the default for debugging scripts.

'`--debug`'
'`-d`' Do not remove the files after a test group was performed —but they are still removed *before*, therefore using this option is sane when running several test groups. Do not create debugging scripts. Do not log (in order to preserve supposedly existing full log file). This is the default for debugging scripts.

'`--trace`'
'`-x`' Trigger shell tracing of the test groups.

12.5 Stand-alone Test Suite

Just like for your first experimentations with Autoconf, it is a good idea to build a tiny test suite independent of all the heavy GNU Build System machinery. Since GNU M4 is an excellent example of a package to exercise, this section is devoted to playing with Autotest on GNU M4. Then the next section, Section 12.6 [Autotesting GNU M4], page 235, taking advantage on our experience, will tackle the problem of embedding the test suite in the whole package.

12.5.1 Simple Uses of Autotest

Just like '`configure.ac`', the very first thing a test suite needs is an identity: the name and version of its hosting package, `AT_PACKAGE_STRING`, an address where failures should be reported, `AT_PACKAGE_BUGREPORT`, and optionally, the test suite's own name, given as argument to the macro `AT_INIT`. When a test suite is embedded in a package, its identity is automatically provided by `configure`, see Section 12.6 [Autotesting GNU M4], page 235; for the time being we will simply `m4_define` them.

Invoking `AT_INIT` is mandatory, as is `AC_INIT` in the Autoconf world:

AT_INIT ([*name*]) [Macro]
Initialize Autotest. Giving a *name* to the test suite is encouraged if your package includes several test suites.

Then, of course, it needs tests. A simple test will suffice for our purpose, for instance checking that `m4` supports the most common options required by the GNU Coding Standards: '`--version`' and '`--help`'.

```
# Process with autom4te to create an -*- Autotest -*- test suite.

m4_define([AT_PACKAGE_STRING],    [GNU Programming 2])
m4_define([AT_PACKAGE_BUGREPORT], [gnuprog2-devel@sourceforge.org])

AT_INIT([Standard Options: 1])

AT_SETUP([Standard Options])
AT_CHECK([m4 --version])
AT_CHECK([m4 --help])
AT_CLEANUP
```

Example 12.6: ‘`std-opt1.at`’ – *An Autotest Source*

This test suite is composed of a single test group, named “Standard Options”. This test group is composed of two steps: checking the reaction of `m4` when given ‘`--version`’ and when given ‘`--help`’.

Test groups are enclosed between `AT_SETUP`/`AT_CLEANUP` pairs:

AT_SETUP (*title*) [Macro]
Begin a test group named *title*. This title is really the identifier of the test group, used in quiet and verbose outputs. It should be short, but descriptive.

AT_CLEANUP [Macro]
End a test group.

To prevent a test group from corrupting another one (via trailing files, modified environment variables and so on), test groups are run by distinct sub-shells in distinct subdirectories. As a direct consequence, test groups cannot share files or variables. To enforce this clean separation between test groups, Autotest ignores anything that is not in a test group. As a consequence, you can run the whole test suite or just some selected test groups in any order without fearing unexpected side effects due to the testing framework itself.

Our unique test group is composed of two steps: ‘`AT_CHECK([m4 --version])`’ stands for “run ‘`m4 --version`’ and expect a success”.

To “compile” this Autotest source file into a Bourne shell-script, run `autom4te`:

```
$ autom4te -l autotest testsuite.at -o testsuite
```

and then run it:

```
## ------------------------------------------------ ##
## GNU Programming 2 test suite: Standard Options: 1. ##
## ------------------------------------------------ ##
  1: std-opt1.at:8     FAILED near ‘std-opt1.at:9’
## ------------------------------------------------ ##
## ERROR: Suite unsuccessful, 1 of 1 tests failed. ##
## ------------------------------------------------ ##

You may investigate any problem if you feel able to do so, in which
case the test suite provides a good starting point.
...
```

Ugh! Something went wrong in our surprisingly simple test suite! The test suite is then re-run verbosely, creating a detailed log file, ‘`std-opt1.log`’, and suggesting sending it to the maintainers.

```
...
Now, failed tests will be executed again, verbosely, and logged
in the file std-opt1.log.
## ------------------------------------------------ ##
## GNU Programming 2 test suite: Standard Options: 1. ##
## ------------------------------------------------ ##
1. std-opt1.at:8: testing Standard Options...
std-opt1.at:9: m4 --version
--- /dev/null   Sat Apr 14 10:11:43 2001
+++ at-stdout   Tue Oct  2 21:33:14 2001
@ -0,0 +1,6 @
+GNU m4 1.4q
+Written by Rene' Seindal and Gary V. Vaughan.
+
+Copyright 1989-1994, 1999, 2000 Free Software Foundation, Inc.
+This is free software; see the source for copying conditions.  There is NO
+warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
1. std-opt1.at:8: FAILED near `std-opt1.at:9'
## ------------------------ ##
## std-opt1.log is created. ##
## ------------------------ ##

Please send `std-opt1.log' to <gnuprog2-devel@sourceforge.org>,
along with all information you think might help.
```

Example 12.7: `std-opt1` *Run*

Reading the verbose output or '`std-opt1.log`' is frightening at first, but with some practice you will soon it find rather easy since it is based on common tools such as `diff`. But let's first spot the failed test group and the guilty test:

```
1. std-opt1.at:8: testing Standard Options...
std-opt1.at:9: m4 --version
--- /dev/null   Sat Apr 14 10:11:43 2001
+++ at-stdout   Tue Oct  2 21:33:14 2001
@ -0,0 +1,6 @
+GNU m4 1.4q
+Written by Rene' Seindal and Gary V. Vaughan.
+
+Copyright 1989-1994, 1999, 2000 Free Software Foundation, Inc.
+This is free software; see the source for copying conditions.  There is NO
+warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
1. std-opt1.at:8: FAILED near `std-opt1.at:9'
```

Each test group is marked at its beginning and its end ('`1. std-opt1.at:8`'). Then each test is presented ('`std-opt1.at:9`'), and, if it failed, some information on the nature of the failure is reported. There are at most three aspects which are checked: the exit status, the standard output, and the standard error output. Here, the unified diff header reports the standard output is not what was expected:

```
--- /dev/null   Sat Apr 14 10:11:43 2001
+++ at-stdout   Tue Oct  2 21:33:14 2001
@ -0,0 +1,6 @
+GNU m4 1.4q
+Written by Rene' Seindal and Gary V. Vaughan.
+
+Copyright 1989-1994, 1999, 2000 Free Software Foundation, Inc.
+This is free software; see the source for copying conditions.  There is NO
+warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

In this report, lines starting with a dash correspond to what was expected, and lines starting with a plus to what was observed. Here, what was observed is

```
GNU m4 1.4q
Written by Rene' Seindal and Gary V. Vaughan.

Copyright 1989-1994, 1999, 2000 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

while the expected content was empty. Indeed our first test, '`AT_CHECK([m4 --version])`', makes no provision for some output. It reads as "run '`m4 --version`', expect a success (exit status should be 0), no standard output, nor standard error output".

Note that no error was reported for the failure of the second test, exercising '`--help`'. This is typical of test groups: it suffices that a single test fails for the rest of the test group to be ignored. The size of test groups is sometimes a matter of taste, but in general a single test group matches a single use scenario. Making long test groups is attractive (less to type etc.) and harmless, but testing independent features should always be done in distinct test groups.

In the present case, '`--version`' and '`--help`' support are definitely two different features: they will be part of different test group. But since they are somewhat related, we will simply keep them close, under the "Standard Options" banner for instance.

We have to fix our tests:

AT_CHECK (*commands*, [*status* = '`0`'], [*stdout*], [*stderr*]) [Macro]
Execute a test by performing given shell *commands*. These commands should normally exit with *status*, while producing expected *stdout* and *stderr* contents.

But what shall we actually expect as standard output? Reproducing the exact result of '`m4 --version`' means your test suite will fail for any other version of GNU M4, similarly with '`--help`'. The easiest is to ignore the standard output, by passing '`ignore`' to `AT_CHECK`:

```
AT_CHECK([m4 --version], [], [ignore])
```

This is what I would have done if I was not writing an Autotest tutorial... Here, we will make sure that *something* is output; and actually, to be even more specific, that both options' output contain '`m4`'. It would certainly be a bug if they didn't...

Autotest does not provide direct support for such content checking[8]. Its model is strictly based on equality, therefore we have to find a means to transform our partial control into a strict equality. There are at least two means to achieve this goal, each having its pros and cons.

The first one that comes to one's mind simply consists in using `grep`, or better yet (more for religious issues than for technically sound reasons), `fgrep`:

[8] Supporting partial checking of contents poses several problems: first it is not an easy task to determine a priori everything the users will need, and second, providing portable support for such features is yet another nightmare that Autotest authors do not want to hear about.

```
AT_CHECK([m4 --version | fgrep m4], [], [ignore])
```

This solution is definitely the simplest, and that's actually its only interest... Imagine the test fails some day: what information will its failure bring?

Almost nothing.

You will not know if `m4` failed, since in portable Bourne shell programming there is no means to determine if an upstream command failed in a pipe: the exit status of the pipe is that of the last command[9]. You will not know either if '`m4 --version`' output a big fat nothing, or nothing that satisfied `fgrep`, since the only output is that of `fgrep`. And last but not least, in the latter case, you won't know *what* was output.

The second solution circumvents these issues simply by decomposing the pipe: first we run '`m4 --version`' *and save its output*, and then we check it:

```
AT_CHECK([m4 --version >m4-version])
AT_CHECK([fgrep m4 m4-version], [], [ignore])
```

Alas, this innocent first test will sooner or later cause you a heart attack: I guarantee someone *will* report problems, most likely an Ultrix user... You just fell into one of the obscure bugs that affect some hosts: they do not support multiple file descriptor redirections. Autotest definitely needs to save the standard output to check its contents, therefore it does redirect both the standard output and the standard error output, hence you must not redirect them again. Equally unfortunate, we know of no way to warn the Autotest user she wrote such an non portable command, we can only ask her to keep it in mind (and anyway, if she doesn't, a fellow Ultrix user will remind her). Note too, that this is why you *must* use `ignore`, and not attempt to redirect the standard output (or error) to '`/dev/null`'.

To address this issue, Autotest may be asked to save the standard output in a file for later examination: use `stdout` to save it into the file '`stdout`':

```
AT_CHECK([m4 --version], [], [stdout])
AT_CHECK([fgrep m4 stdout], [], [ignore])
```

The case of the '`--help`' is exactly similar, so much indeed that writing a macro would factor a lot of code.

12.5.2 Writing Autotest Macros

Autotest provides a minimal set of macros, flexible enough to meet the needs, but too limited to perform strict tests on some executables. This is on purpose: there is no universal magical way to check an executable (or if you have one, please contact the Autotest maintainers as soon as possible). You will have to write your testing tools, i.e., since we are describing an M4 based programming environment, you will have to write your own test macros.

We will write a macro which to factor a whole test group exercising `m4` on '`--version`' and '`--help`', and as usual, the most difficult task will be finding a good name for it. I suggest `AT_CHECK_M4_STD_OPTION` (users willing to contribute better names are most welcome: send submissions to `gnuprog2-devel@sourceforge.org`, along with all information you think might help). The whole '`std-opt2.at`' now contains:

[9] Modern shells no longer have this deficiency, for instance with Zsh:

```
$ false | cat >/dev/null; echo "$?; $pipestatus[1]; $pipestatus[2]"
0; 1; 0
```

```
# Process with autom4te to create an -*- Autotest -*- test suite.

m4_define([AT_PACKAGE_STRING],    [GNU Programming 2])
m4_define([AT_PACKAGE_BUGREPORT], [gnuprog2-devel@sourceforge.org])

AT_INIT([Standard Options: 2])
AT_BANNER([Standard Options.])

# AT_CHECK_M4_STD_OPTION(OPTION)
# ------------------------------
# Check that `m4 OPTION' outputs something containing `m4'.
m4_define([AT_CHECK_M4_STD_OPTION],
[AT_SETUP([$1 support])
AT_CHECK([m4 $1], [], [stdout])
AT_CHECK([fgrep m4 stdout], [], [ignore])
AT_CLEANUP])
AT_CHECK_M4_STD_OPTION([--version])
AT_CHECK_M4_STD_OPTION([--help])
```

Example 12.8: *'std-opt2.at' – An Autotest Source*

which gives:

```
## ------------------------------------------------ ##
## GNU Programming 2 test suite: Standard Options: 2. ##
## ------------------------------------------------ ##

Standard Options.

  1: std-opt2.at:19    ok
  2: std-opt2.at:20    ok
## ---------------------------- ##
## All 2 tests were successful. ##
## ---------------------------- ##
```

Example 12.9: std-opt2 *Run*

Tada!

Let's go a step further: after all, these two test groups are fairly generic: we could very well introduce a higher level macro to check whether some program supports '--version' and '--help'! We will exercise autoconf, gcc, litbool, and m4.

```
# Process with autom4te to create an -*- Autotest -*- test suite.

m4_define([AT_PACKAGE_STRING],    [GNU Programming 2])
m4_define([AT_PACKAGE_BUGREPORT], [gnuprog2-devel@sourceforge.org])

AT_INIT([Standard Options: 3])
```

```
# _AT_CHECK_STD_OPTION(PROGRAM, OPTION)
# -------------------------------------
# Check that 'PROGRAM OPTION' outputs something containing 'PROGRAM'.
m4_define([_AT_CHECK_STD_OPTION],
[AT_SETUP([$1 $2 support])
AT_CHECK([$1 $2], [], [stdout])
AT_CHECK([fgrep $1 stdout], [], [ignore])
AT_CLEANUP])

# AT_CHECK_STD_OPTIONS(PROGRAM)
# -----------------------------
# Check that PROGRAM respects the GCS wrt --version, and --help.
m4_define([AT_CHECK_STD_OPTIONS],
[AT_BANNER([$1 Standard Options.])
AT_TESTED([$1])
_AT_CHECK_STD_OPTION([$1], [--version])
_AT_CHECK_STD_OPTION([$1], [--help])
])

AT_CHECK_STD_OPTIONS([autoconf])
AT_CHECK_STD_OPTIONS([gcc])
AT_CHECK_STD_OPTIONS([libtool])
AT_CHECK_STD_OPTIONS([m4])
```

Example 12.10: 'std-opt3.at' – *An Autotest Source*

which produces:

```
## ------------------------------------------------ ##
## GNU Programming 2 test suite: Standard Options: 3. ##
## ------------------------------------------------ ##

autoconf Standard Options.

  1: std-opt3.at:27    ok
  2: std-opt3.at:27    ok

gcc Standard Options.

  3: std-opt3.at:28    FAILED near 'std-opt3.at:28'
  4: std-opt3.at:28    ok

libtool Standard Options.

  5: std-opt3.at:29    ok
  6: std-opt3.at:29    ok

m4 Standard Options.

  7: std-opt3.at:30    ok
  8: std-opt3.at:30    ok

## ------------------------------------------- ##
## ERROR: Suite unsuccessful, 1 of 8 tests failed. ##
## ------------------------------------------- ##
```

```
You may investigate any problem if you feel able to do so, in which
case the test suite provides a good starting point.

Now, failed tests will be executed again, verbosely, and logged
in the file std-opt3.log.
## ------------------------------------------------ ##
## GNU Programming 2 test suite: Standard Options: 3. ##
## ------------------------------------------------ ##
3. std-opt3.at:28: testing gcc --version support...
std-opt3.at:28: gcc --version
stdout:
2.95.2
std-opt3.at:28: fgrep gcc stdout
stdout:
std-opt3.at:28: exit code was 1, expected 0
3. std-opt3.at:28: FAILED near `std-opt3.at:28'
## ------------------------ ##
## std-opt3.log is created. ##
## ------------------------ ##

Please send `std-opt3.log' to <gnuprog2-devel@sourceforge.org>,
along with all information you think might help.
```

Example 12.11: `std-opt3` *Run*

Please note that we clearly achieved our goal thanks to the two step test: we *know* exactly why it failed with `gcc`, since the whole output is displayed in the logs, there is no need for additional interaction with the user, had the failure occurred on such a "remote" environment.

12.5.3 Checking `dnl` and `define`

Now that our snacks have been digested, let's focus again on our main goal: designing a GNU M4 test suite, i.e., exercising the real core features of M4. We will write simple tests for a few basic builtins, and we will be sure to exercise them in some invalid way (see Section 12.2.3 [Look for Realism], page 212).

Specifying the identity of the hosting package is also painful, in particular because a proper definition of the `AT_PACKAGE_STRING` must include its version. To factor the definition of these variables from now we will rely on the existence of '`package.m4`':

```
# Signature of the current package.
m4_define([AT_PACKAGE_STRING],    [GNU Programming 2E])
m4_define([AT_PACKAGE_BUGREPORT], [gnuprog2-devel@sourceforge.org])
```

Example 12.12: *A Simple* '`package.m4`'

You don't have to `m4_include` it, or pass it as an argument to `autom4te`: '`autom4te --language=autotest`' automatically includes this file if present.

Two candidates are already laid in the test bed: `dnl`, being the most commonly used builtin[10], and `define`, so that we can test the user macro expansion. Exercising `m4` typically consists in

[10] The Fileutils' '`configure.ac`' invokes `dnl` 198920 times, followed by `shift` (119196), `ifdef` (88623). The most used Autoconf macro was `AC_PROVIDE`, with a miserable score of 4202. While YMMV, be sure that `AC_INIT` was invoked once.

running it on some files. You are absolutely free to create the files the way you want, for instance using `AT_DATA`.

AT_DATA (*file, contents*) [Macro]

Initialize an input data *file* with the given *contents*. Of course, the *contents* have to be properly quoted between square brackets to protect against included commas or spurious `m4` expansion; no shell expansion of any sort is performed. The contents ought to end with an end of line.

Testing `dnl` is exceedingly simple: give it something to swallow, and observe it did:

```
# Process with autom4te to create an -*- Autotest -*- test suite.
# dnl.at -- Testing GNU M4 'dnl' and 'define' builtins.

AT_INIT([m4])


AT_SETUP([Dnl])
AT_DATA([[input.m4]],
[[dnl This is killed.
This is not
]])
AT_TEST([[m4 input.m4]], [],
[[This is not
]])
AT_CLEANUP
```

Example 12.13: *'`dnl.at`' (i) – A Broken Autotest Source Exercising* `dnl`

```
$ autom4te -l autotest dnl.at -o dnl
dnl.at:8: error: possibly undefined macro: dnl
dnl.at:11: error: possibly undefined macro: AT_TEST
$
```

Arg! Yet another zealous useful feature: `autom4te` makes sure there are no suspicious tokens in the output which could result from improper quotation, or typing errors. And there is one indeed: the author of the test suite meant `AT_CHECK`, not `AT_TEST`. But in this test suite we really want to refer to `dnl`.

There are two means to explain this to `autom4te`.

One first solution consists in using the empty quadrigraph, '`@&t@`', to mark valid occurrences of `dnl` in the output, as in:

```
AT_DATA([[input.m4]],
[[d@&t@nl This is killed.
This is not
]])
```

But `autom4te` still complains, this time, being unable to find the source of the guilty `dnl` in '`dnl.at`', its input, it reports the location in the output file:

```
$ autom4te -l autotest dnl.at -o dnl
dnl:209: error: possibly undefined macro: dnl
$ sed -n 209p dnl
   1: dnl.at:6        Dnl
```

Aha! The culprit is no less than the filename! Therefore we have no other choice than using the second solution (except renaming the file): completely disabling the checking of `dnl` in

the output. As matter of fact, we will use `dnl` so heavily while testing `m4` that tagging each occurrence would obfuscate too much: just add '`m4_pattern_allow([^dnl$])`'[11].

The first solution is definitely the safest, because you tagged exactly the occurrences of `dnl` which are meant to be output. Any other accidental unexpanded `dnl` will still be caught. But sometimes simplicity and risks are to be preferred to strictness and safety.

And now for something completely different: `define`. Contrary to `dnl`, `define` has a precise arity: without arguments it is ignored, otherwise it takes one or two arguments, it should warn for any other arity (obviously we won't test them all). Let's first check by hand:

```
$ cat define.m4
define
define()
define(‘one’)one
define(‘two’, ‘Two’)two
define(‘three’, ‘Three’, ‘THREE’)three
$ m4 define.m4
define


Two
error m4: define.m4: 5: Warning: define: too many arguments (ignored): 3 > 2
Three
```

It is then a simple matter of separating the standard output from the standard error output, and just wrap this into a test case:

```
AT_SETUP([[Define]])

AT_DATA([[define.m4]],
[[define
define()
define(‘one’)one
define(‘two’, ‘Two’)two
define(‘three’, ‘Three’, ‘THREE’)three
]])
AT_CHECK([[m4 define.m4]], [],
[[define


Two
Three
]],
[[m4: define.m4: 5: Warning: define: too many arguments (ignored): 3 > 2
]])

AT_CLEANUP
```

Example 12.14: *'`dnl.at`' (ii) – An Autotest Source Exercising* `define`

Create the test suite, launch it: good, it passes with success. Let's try it on our fetal `m4`, not yet installed:

[11] You might have considered '`m4_pattern_allow([^dnl\.at$])`', but this won't work since the output is split into words, and here there are two: '`dnl`' and '`at`'.

```
$ ./dnl AUTOTEST_PATH=$HOME/src/m4/src
## ------------------------------------------------ ##
## GNU Programming 2E 0.0a test suite: Dnl and Define. ##
## ------------------------------------------------ ##
  1: dnl.at:7          ok
  2: dnl.at:17         FAILED near `dnl.at:35'
## ---------------------------------------------- ##
## ERROR: Suite unsuccessful, 1 of 2 tests failed. ##
## ---------------------------------------------- ##

You may investigate any problem if you feel able to do so, in which
case the test suite provides a good starting point.

Now, failed tests will be executed again, verbosely, and logged
in the file dnl.log.

## ------------------------------------------------ ##
## GNU Programming 2E 0.0a test suite: Dnl and Define. ##
## ------------------------------------------------ ##
2. ./dnl.at:17: testing Define...
./dnl.at:35: m4 define.m4
--- -   Tue Sep  4 18:04:30 2001
+++ at-stderr   Tue Sep  4 18:04:30 2001
@ -1,2 +1,2 @
-m4: define.m4: 5: Warning: define: too many arguments (ignored): 3 > 2
+lt-m4: define.m4: 5: Warning: define: too many arguments (ignored): 3 > 2

2. ./dnl.at:17: FAILED near `dnl.at:35'
## ------------------ ##
## dnl.log is created. ##
## ------------------ ##

Please send `dnl.log' to <gnuprog2-devel@sourceforge.org>,
along with all information you think might help.
```

Example 12.15: `dnl` *Run on an Installed* `m4`

Because it is the paragon of dynamic module based software, GNU M4 is built with Libtool; because of obscure but very well founded reasons which are beyond the scope of this chapter (**FIXME:** *Ref to Libtool?.*), '`bin/m4`' is actually a shell script. It runs an executable named `lt-m4`. This is why the signature in the error message is "wrong". We have a serious problem, for our ultimate goal is to write a test suite shipped with GNU M4.

One possibility consists in adjusting the expected error messages to using '`lt-m4`'. This would prevent us from using our test suite on any other `m4`. In addition it clashes with an important motto: the test suite is a user, see Section 12.2.3 [Look for Realism], page 212.

Another is having the test suite be robust enough to work with different signatures, i.e., applying the same techniques as those we used in the previous section: save the standard error output, standardize it, check it. What a hassle! But why not, an `AT_CHECK_M4` macro could hide those gory details.

For the time being, let us just imagine we didn't read Section 12.2 [Designing a Test Suite], page 211. We chose this solution, and proceed to other kinds of tests.

12.5.4 Checking Module Support

A major aspect of GNU M4 is its wonderful handling of modules. As a matter of fact, the executable `m4` is nothing but an empty shell which sole ability is almost reduced to handling '`--help`' and '`--version`' (which we already exhaustively tortured). Such a major feature must be exercised, and in fact, any conscientious maintainer will take a sadist pleasure at writing the most perverted possible tests. Gary is one such person, and we will follow his tracks, checking that modules can be loaded and unloaded.

Our victim will be the `gnu` module, which contains both builtins, such as `builtin`, and macros, such as `__gnu__`. We will check that they are undefined at start up when '`--traditional`' is specified, defined when `gnu` is loaded, and undefined again when the module is unloaded, and defined when loaded again:

```
# Process with autom4te to create an -*- Autotest -*- test suite.
# modules.at -- Testing GNU M4 module support.

AT_INIT([Modules support])

AT_SETUP([Modules loading and unloading])

AT_DATA([[input.m4]],
[[define(‘status’,
‘‘$1’: ifdef(‘$1’, ‘defined’, ‘non defined’)’)
status(‘builtin’), status(‘__gnu__’)
load(‘gnu’)status(‘builtin’), status(‘__gnu__’)
unload(‘gnu’)status(‘builtin’), status(‘__gnu__’)
load(‘gnu’)status(‘builtin’), status(‘__gnu__’)
]])

AT_CHECK([[m4 --traditional --load-module=load input.m4]], [],
[[
builtin: non defined, __gnu__: non defined
builtin: defined, __gnu__: defined
builtin: non defined, __gnu__: non defined
builtin: defined, __gnu__: defined
]])

AT_CLEANUP
```

Example 12.16: '`modules.at`' – *An Autotest Source Checking M4 Modules Support*

which indeed runs as expected: 100% of one test passes. Note however that this test is actually quite weak, with some more effort it would have been better to check that the functionalities of `builtin` and `__gnu__` are still working.

Now, again, we can run the test suite on our working copy of `m4`, by a simple '`./modules AUTOTEST_PATH=$HOME/src/m4/src`'. But observe that if you replace '`m4`' with '`strace m4`' or something equivalent, and ask for the standard error to be '`ignore`''d, then you get something similar to:

```
$ ./modules -v AUTOTEST_PATH=$HOME/src/m4/src | grep gnu
open("/home/akim/src/m4/modules/.libs/gnu.so.0", O_RDONLY) = 3
open("/usr/local/libexec/m4/gnu.la", O_RDONLY) = 4
read(4, "# gnu.la - a libtool library fil"..., 4096) = 708
open("/usr/local/libexec/m4/gnu.la", O_RDONLY) = 4
read(4, "# gnu.la - a libtool library fil"..., 4096) = 708
```

Although '`/home/akim/src/m4/modules/.libs/gnu.so.0`' is reassuring, since it demonstrates that Libtool took care of loading the non installed version of the `gnu` module, the '`/usr/local/libexec/m4/gnu.la`' part is still a bit frightening: what if, after all, we were mixing installed modules with a non installed `m4`? As a matter of fact, this problem is extremely frequent since today many executables use auxiliary files. For instance the Autoconf collection heavily depends on a configuration file named '`autom4te.cfg`' and on many M4 files, Bison and Flex need to find "skeleton" files[12], Automake needs scripts like '`install-sh`' and '`missing`', Makefile components '`*.am`' etc. When testing the working copy of these tools the risk of mixing installed and non installed bits is high, and will of course result in insignificant results, whether the test suite passes or not.

The most common answer is having the test suite pass some combinations of options and environment variables to make sure the tools load non installed files. In the current case, it means replacing the previous `AT_CHECK` invocation with something like

```
AT_CHECK([[m4 -G -M $HOME/src/m4/modules -m load input.m4]], ...
```

Now we have the converse problem: the test suite will always involve non installed modules, even when exercising an installed `m4`.

The easy answer to this dilemma is: "forget about testing an installed program, after all it should have been tested *before* being installed", in other words "forget about testing any other copy of the program than the one in the same *build* tree as this test suite".

It is worth mentioning the case of programs invoking other programs in the same package. Autoconf is a typical example: `autom4te` is slaved by `autoconf`, `autoheader`, `autoscan` and `autoupdate`, all of them being run by `autoreconf`! Yet this is an improvement over the previous situation where, for instance, `autoheader` ran `autoconf`, itself using `autom4te`. In order to enforce this relationship, all of them had hard coded heuristics like this:

```
# Default AUTOCONF to the name under which 'autoconf' is installed
# when './configure --program-transform-name' and similar is used.
: ${AUTOCONF=@autoconf-name@}
dir='echo "$0" | sed -e 's,[^/]*$,,''
# We test "$dir/autoconf" in case we are in the build tree,
# in which case the names are not transformed yet.
for autoconf in "$AUTOCONF" \
                "$dir/@autoconf-name@" \
                "$dir/autoconf" \
                "@bindir@/@autoconf-name@"; do
  test -f "$autoconf" && break
done
```

Example 12.17: *Excerpt of* `autoheader` *Looking for* `autoconf`

Let's list a few consequences:

– Because directories are hard coded, if you move something, the behavior is undefined.

– The innocent '`test -f "$autoconf"`' is hell! If the user specified some option '`--foo`' in `AUTOCONF`, then this snippet will look for the file '`autoconf --foo`'.

– If the user specified '`AUTOCONF=autoconf-2.13`', then again it won't be honored since instead of letting the system look for it in the `PATH`, this snippet just checks if `autoconf-2.13` is present in the *current* directory.

[12] Bison and Flex both generate tables describing the grammar specificities, but the engine, the code which "executes" these tables is independent of the grammar. This code, instead of being hard coded in the executables, is stored in a file named the *skeleton*. (**FIXME:** *Didier says this footnote should be removed. Once Bison and Flex documented, ref to there.*.)

– User specifications are silently ignored! In neither of the previous cases the user will be warned of what happened.

The explosion of the European Launcher, see Section 12.1.3 [Ariane 501], page 210, caused by a $500 000 000 bug, was caused by a comparable kind of problem: the real application was polluted with code unrelated to a normal use. Today, the Autoconf programs simply include:

```
: ${AUTOCONF=@autoconf-name@}
```

and *that's all*! Let `PATH` handle the rest: when being tested in the build tree, the wrappers are run and handle all the dark magic. Now the behavior, relying on standard Unix interface, is predictable, and will properly fail when it should.

It is a mistake to dedicate a test suite to the special layout of a package in the process of being built. We strongly discourage you from going to the dark side of the testing force.

So, we are back to our problem: how can we test both an installed program, and an non installed program while having each copy use its own files? Keep in mind that *a test suite must be seen as a user*, albeit eccentric and demanding: present the same interface in both cases to the test suite. Then the answer is obvious: provide a wrapper, a small shell script which takes care of running a non installed program, and let the `PATH` handle the rest.

This wrapper must give a perfect illusion, it must pass the Turing test: an observer shouldn't be able to tell the difference. Note that we also just solved the problem left in the previous section —'`lt-m4`' vs. '`m4`' in error messages—: this wrapper must hide this detail. Since this wrapper depends upon configuration options, it is `configure` which will instantiate from a template.

In the case of `m4`, this template, '`m4.in`', is just:

```
#! /bin/sh
# @configure_input@
# Wrapper around a non installed m4 to make it work as an installed one.

"@top_buildpath@/src/m4" \
      --module-directory="@top_buildpath@/modules" \
      ${1+"$@"} 2>/tmp/m4-$$
status=$?
# Normalize stderr.
sed 's,^[^:]*[lt-]*m4[.ex]*:,m4:,' /tmp/m4-$$ >&2
rm /tmp/m4-$$

exit $status
```

Example 12.18: '`m4.in`' – *A Wrapper around a non installed* `m4`

How does this script work? We pass the option '`--module-directory`' so that it uses the non installed modules instead of those possibly installed on the system; and we used `sed` to normalize the name of the executable in the error messages. This sed invocation deserves some explanations:

'`[^:]*`' aims at removing the possible leading path. In particular, when configured with '`--disable-shared`', '`@top_buildpath@/src/m4`' is an genuine binary, which will display its full path.

'`[lt-]*`' is a portable approximation of '`\(lt-\)\?`', matching Libtool's prefix when not configured with '`--disable-shared`'. In this case, there is no real need to normalize a possible directory specification because the '`bin/m4`' wrapper modifies the `PATH` to run the actual executable, in which case the name is indeed simply '`m4`'.

‘[.ex]*’ takes care of the possible ‘.exe’ extension on some poor hosts.

Approximating even further, for instance with ‘s/^[^:]*:/m4:/’, will sooner or later destroy other standard error output than m4’s signature, the output of dumpdef for instance. One could also rely on a redefinition of PATH in which case the normalization can be simplified.

Note that these wrappers are also a good place where special magical tricks can be performed. For instance, as described in Section 12.2.7 [Other Uses of a Test Suite], page 216, the test suite can be a good place for profiling. Configure the package using ‘./configure CFLAGS=’-pg’’ (and ‘--disable-shared’ if, as is the case of GNU M4, the application is heavily composed of libraries), and add a few lines to save the profiling data file, ‘gmon.out’, for a later use (see Chapter 15 [Profiling and Optimising Your Code], page 263):

```
#! /bin/sh
# @configure_input@
# Wrapper around a non installed m4 to make it work as an installed one.

"@top_buildpath@/src/m4" \
      --module-directory="@top_buildpath@/modules" \
      ${1+"$@"} 2>/tmp/m4-$$
status=$?
test -d gmon || mkdir gmon
mv gmon.out gmon/$$
# Normalize stderr.
sed 's,^[^:]*[lt-]*m4[.ex]*:,m4:,' /tmp/m4-$$ >&2
rm /tmp/m4-$$

exit $status
```

Example 12.19: *‘m4.in’ – A Profiling Wrapper*

then run the test suite, then ‘gprof -s ../src/m4 gmon/* && rm -rf gmon’, and finally ‘gprof ../src/m4 gmon.sum’. Enjoy!

Well, there is not much to enjoy, because the GNU M4 test suite is really paying attention to testing independent features, and includes almost no torture tests. But applying the same trick on an M4 based package, such as Autoconf (i.e., installing the wrapper above as ‘tests/m4’ in Autoconf’s ‘tests’ directory) provides a excellent base for profile-guided improvements.

12.5.5 Testing Optional Features

Some packages provide optional features, possibly depending upon configuration options. Therefore a test suite will exercise programs with different behaviors at runtime, or put more simply, it will test different programs sharing the same name. Quite by a mere chance, GNU M4 is one such program: depending upon the system and/or the user’s will, support for extended precision arithmetics based on GMP[13] might be compiled.

Therefore we need optional tests. The special exit status value ‘77’ tells AT_CHECK to disregard the rest of the test group, independent of the results (unless the test expects 77 as exit value, see (**FIXME:** *Inside Test Groups.*), AT_CHECK, for more details).

Autotest based test suites provide the user with a means to pass configuration options to the tests: ‘atlocal’. The file ‘atlocal’ is automatically generated from its template, ‘atlocal.in’,

[13] GMP, the GNU Multiple Precision arithmetic library, is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. It strives to provide the fastest possible arithmetic for all applications that need higher precision than is directly supported by the basic C types.

by AC_CONFIG_TESTDIR ((**FIXME:** *Embedding an Autotest Test Suite.*)), and is loaded (when present) by any Autotest test suite. In our case, we merely need to know if GMP support was compiled in, which configure knows via the value of the variable USE_GMP: either 'yes' or 'no'. Therefore, our template 'atlocal.in' is:

```
# -*- shell-script -*-
# @configure_input@
# Configurable variable values for M4 test suite.
# Copyright 2000, 2001 Free Software Foundation, Inc.

# Some tests cannot be performed with all the configurations.
USE_GMP=@USE_GMP@
```

Example 12.20: 'tests/atlocal.in' – *User Test Variables*

There remains to write the test itself, exercising the mpeval module:

```
# Process with autom4te to create an -*- Autotest -*- test suite.
# mpeval1.at -- Testing GNU M4 GMP support.

AT_INIT([GMP support])

AT_SETUP([GMP: Exponentiation])

AT_CHECK([test "$USE_GMP" = yes || exit 77])

AT_DATA([[input.m4]],
[[mpeval(‘2**100’)
]])

AT_CHECK([[m4 -m mpeval input.m4]], 0,
[[1267650600228229401496703205376
]])

AT_CLEANUP
```

Example 12.21: 'mpeval1.at' – *Exercising an Optional Feature Using* 'atlocal'

create mpeval1, and run it:

```
## ---------------------------------------------- ##
## GNU Programming 2E 0.0a test suite: GMP support. ##
## ---------------------------------------------- ##
  1: mpeval.at:6       ok (skipped near ‘mpeval.at:8’)
## ---------------------------------------- ##
## All 1 tests were successful (1 skipped). ##
## ---------------------------------------- ##
```

Hm... Something went wrong.... I'm sure the m4 I just installed *has* GMP support...

If you spend some time analyzing the failure, you'll find a simple explanation: the example is presented as it if were part of GNU M4 distribution, while in fact it's part of a different package, gnu-prog2. The latter has no information with respect to the configuration options used for the GNU M4 which was installed!

The problem we face is well known to Autoconf gurus: differences between *configure time* data (not very different from compile time data), and *execution time* data. Some softwares make decision with regards to their behavior at *runtime*, e.g., some hosts can be big endian or little endian depending on runtime environment. In such a case, you should just follow the program's footsteps and adjust your tests to runtime conditions. As a matter of fact, still following the

motto "the test suite is a user" (see Section 12.2.3 [Look for Realism], page 212), you should depend as little as possible on configuration options: a user has no reason to know the decision made by her system administrator. In the case of GMP, it means first asking `m4` whether it knows a module named `mpeval`, and then checking it:

```
# Process with autom4te to create an -*- Autotest -*- test suite.
# mpeval2.at -- Testing GNU M4 GMP support.

AT_INIT([GMP support: Runtime Check])

AT_SETUP([GMP: Exponentiation])

AT_CHECK([m4 -m mpeval </dev/null || exit 77])

AT_DATA([[input.m4]],
[[mpeval(‘2**100’)
]])

AT_CHECK([[m4 -m mpeval input.m4]], 0,
[[1267650600228229401496703205376
]])

AT_CLEANUP
```

Example 12.22: ‘`mpeval2.at`’ – *Exercising an Optional Feature at Runtime*

This time, the `mpeval` module is properly exercised:

```
## ------------------------------------------------------------ ##
## GNU Programming 2E 0.0a test suite: GMP support: Runtime check. ##
## ------------------------------------------------------------ ##
  1: mpeval.at:6
## ---------------------------- ##
## All 1 tests were successful. ##
## ---------------------------- ##
```

Creating stand-alone test suites is still rare, and hackers writing test suites for other people's packages are even less common[14]: the most common use of Autotest is writing a portable test suite shipped with the tested package, within the GNU Build System, together with Automake and Autoconf. It turns out miraculously to be the topic of the next section, Section 12.6 [Autotesting GNU M4], page 235.

12.6 Autotesting GNU M4

In this section, as a demonstration of the principles presented in the previous sections, we explore the main steps followed in the design of the GNU M4 test suite. As a matter of fact GNU M4 already had a test suite composed of tests generated from the documentation, and a set of small shell scripts implementing test groups. Hence, its Autotestification merely consisted in re-engineering it.

[14] People performing torture tests to `telnet`, `su` etc. are named *crackers*, not *hackers*, and therefore they do not invalidate my sentence.

12.6.1 The GNU M4 Test Suite

Based on the general advice presented in Section 12.2.4 [Ordering the Tests], page 213, the first steps consisted in determining the sequence of kinds of tests to be performed: (i) exercising macro and builtin definitions and uses, (ii) builtins (‘`builtins.at`’, (iii) special options (‘`options.at`’, (iv) complex hand crafted tests (‘`others.at`’), (v) module support (‘`modules.at`’, and (vi) generated tests, automatically extracted from the documentation (‘`generated.m4`’).

Because we repeatedly run `m4` with some common options, we also define two macros which will make our life easier: `AT_CHECK_M4` which is a simple check, and `AT_TEST_M4`, which is a whole test group in itself. As advised in Section 12.2.3 [Look for Realism], page 212, for options such as ‘`--warning`’, we pass ‘`-d`’, ‘`--debug`’, to constantly check the debugging output (which is actually produced only when special macros such as `traceon` are invoked). We also pass the option ‘`-b`’, ‘`--batch`’, to make sure the test suite is interruptible. This point deserves some slightly out off-topic detailed explanations, typically a footnote, but that too lengthy to fit down there...

> Interactive programs, such as shells, often check whether their standard input is a TTY (basically meaning that the standard input is the user herself, not a file), and then neutralize *CTRL-c*, since it would result in exiting the shell. GNU M4 follows the same rule, and therefore, if you run ‘`m4`’, typing *CTRL-c* will have no effect, while running ‘`m4 <input.m4`’ keeps *CTRL-c* activated. But what happens when running ‘`m4 input.m4`’? The standard input is *not* ‘`input.m4`’, the latter is just an argument passed to `m4` on its command line, but the standard input is not redirected, and therefore it remains the same as before (typically, users run `./testsuite` from an interactive shell, hence the standard input of `testsuite` *is* a TTY, inherited by `m4`, therefore `m4` considers it is in interactive mode!).
>
> If for some reason ‘`input.m4`’ makes `m4` go into in infinite loop, then you are doomed, you will have to use `kill` to terminate the process. It should be noted that other interactive programs are still sensible to *CTRL-c*: they stop the current operation and resume to the prompt. GNU M4 has no such feature, which makes it even worse. I strongly encourage using systematically ‘`--batch`’.
>
> Finally, please note that sometimes, for some reason, a test might not behave as expected and may be expecting some input from the standard input. Then the test suite will appear to be stuck. If you experience this, if some user reports a never ending test group, suggest that they run ‘`./testsuite </dev/null`’. If this time the test group ends, ask her to run ‘`./testsuite -x`’: the last command was the one expecting data from the standard input.

Example 12.23: *Testing Interactive Programs*

Putting all this together gives the following ‘`testsuite.at`’. To save trees, the license is not included below.

```
# Process with autom4te to create an -*- Autotest -*- test suite.

# Test suite for GNU M4.
# Copyright 2001 Free Software Foundation, Inc.

# We need at least Autotest 2.52g, otherwise fail now.
m4_version_prereq([2.52g])
```

```
# AT_CHECK_M4(ARGS, [EXIT-STATUS = 0], [STDOUT = ''], [STDERR = ''])
# ------------------------------------------------------------------
m4_define([AT_CHECK_M4],
[AT_CHECK([m4 -b -d $1], [$2], [$3], [$4])
])

# AT_TEST_M4(TITLE, INPUT, [STDOUT = ''], [STDERR = ''])
# ------------------------------------------------------
# Run m4 on INPUT, expecting a success.
m4_define([AT_TEST_M4],
[AT_SETUP([$1])
AT_DATA([[input.m4]], [$2])
AT_CHECK_M4([[input.m4]], 0, [$3], [$4])
AT_CLEANUP
])

# We use 'dnl' in zillions of places...
m4_pattern_allow([^dnl$])

# We exercise m4.
AT_TESTED([m4])

## ----------- ##
## The suite.  ##
## ----------- ##

AT_INIT

# Macro definitions, uses, tracing etc.
m4_include([macros.at])

# Torturing builtins.
m4_include([builtins.at])

# Options.
m4_include([options.at])

# Hand crafted tests.
m4_include([others.at])

# Torturing the modules support.
m4_include([modules.at])

# From the documention.
m4_include([generated.at])
```

Example 12.24: GNU *M4's* `testsuite.at`

Most tests are straightforward and do not deserve special attention; to see `AT_CHECK_M4` and `AT_TEST_M4` in action, see the GNU M4 distribution. We will focus on excerpts of '`modules.at`' and '`generated.at`'.

Originally, when the test suite was only a set of handwritten shell scripts, a few of them were testing the loading and unloading of modules, sometimes testing relative path to modules, sometimes absolute paths, some were exercising the option '`--module-directory`', others the environment variable `M4MODPATH`, and others `LTDL_LIBRARY_PATH`. Looking at this set of shell scripts it was barely possible to verify their coverage: were there cases which were not tested? In addition, did all the tests have the same strength (the inputs were sometimes different)? As

advised in Section 12.2.6 [Maintain the Test Suite], page 215, these specific tests were generalized into a unique test group macro, and it then became easy to be sure all the possibilities were covered. Since in addition these tests depend on `modtest`, a module written specially for exercising the modules, and therefore which is not to be installed, the macro is equipped with a preliminary test to skip the test group when it missing. Please note that since these tests aim at checking that `modtest` can be *found*, using '`AT_CHECK([m4 -m modtest.la || exit 77])`' is taking the risk that actual failures be considered as skipped tests.

```
## -------------------------- ##
## Exercising the test module.  ##
## -------------------------- ##

# AT_TEST_M4_MODTEST(TITLE, ENV-VARS, M4-OPTIONS)
# -----------------------------------------------
# Skip if modtest is not present (we are not in the package).
m4_define([AT_TEST_M4_MODTEST],
[AT_SETUP([$1])
AT_KEYWORDS([module])

AT_CHECK([test -f $top_builddir/modules/modtest.la || exit 77])

AT_DATA([input.m4],
[[load(‘modtest’)
test
Dumpdef: dumpdef(‘test’).
unload(‘modtest’)
test
Dumpdef: dumpdef(‘test’).
]])

AT_CHECK([$2 m4 -m load -d input.m4 $3], 0,
[[
Test module called.
Dumpdef: .

test
Dumpdef: .
]],
[[Test module loaded.
test:   <test>
Test module unloaded.
m4: input.m4: 6: Warning: dumpdef: undefined name: test
]])

AT_CLEANUP
])

AT_TEST_M4_MODTEST([--module-directory: absolute path],
                   [], [-M $top_buildpath/modules])

AT_TEST_M4_MODTEST([--module-directory: relative path],
                   [], [-M $top_builddir/modules])

AT_TEST_M4_MODTEST([M4MODPATH: absolute path],
                   [M4MODPATH=$top_buildpath/modules], [])
```

```
AT_TEST_M4_MODTEST([M4MODPATH: relative path],
                   [M4MODPATH=$top_builddir/modules], [])
AT_TEST_M4_MODTEST([LTDL_LIBRARY_PATH: absolute path],
                   [LTDL_LIBRARY_PATH=$top_buildpath/modules], [])
AT_TEST_M4_MODTEST([LTDL_LIBRARY_PATH: relative path],
                   [LTDL_LIBRARY_PATH=$top_builddir/modules], [])
```

The last bit of testing we will pay attention to is the case of the tests extracted from the documentation. The file '`generated.at`' is produced by a simple Awk program, '`generate.awk`', which we won't detail here, see the GNU M4 distribution. The idea is simple: convert the example from the Texinfo documentation into actual tests. For instance, the following excerpt of the node "Dumpdef" of '`m4.texinfo`' (see section "Displaying macro definitions" in *GNU m4 – A powerful macro processor*):

```
@example
define(‘foo’, ‘Hello world.’)
@result{}
dumpdef(‘foo’)
@error{}foo:  ‘Hello world.’
@result{}
dumpdef(‘define’)
@error{}define:  <define>
@result{}
@end example
```

rendered as

```
define(‘foo’, ‘Hello world.’)
⇒
dumpdef(‘foo’)
error foo:  ‘Hello world.’
⇒
dumpdef(‘define’)
error define:  <define>
⇒
```

is turned into:

```
## --------- ##
## Dumpdef.  ##
## --------- ##

AT_SETUP([[Dumpdef]])
AT_KEYWORDS([[documentation]])

# ../doc/m4.texinfo:1673
AT_DATA([[input.m4]],
[[define(‘foo’, ‘Hello world.’)
dumpdef(‘foo’)
dumpdef(‘define’)
]])
```

```
AT_CHECK_M4([[input.m4]], 0,
[[


]],
[[foo:  'Hello world.'
define:  <define>
]])
AT_CLEANUP
```

12.6.2 Using Autotest with the GNU Build System

As far as Autoconf is concerned, you only have to invoke `AC_CONFIG_TESTDIR`:

AC_CONFIG_TESTDIR (*test-directory*, [*autotest-path* = *test-directory*]) [Macro]
Ask for the creation of '*test-directory*`/atconfig`', which contains Autotest private information related to the layout of the package tree.

The test suite default path, `AUTOTEST_PATH`, is set to *autotest-path*. This colon-separated path should include the directories of the programs to exercise, relative to the top level of the package.

and create the wrapper around `m4`:

```
AC_CONFIG_TESTDIR(tests)
AC_CONFIG_FILES([tests/m4], [chmod +x tests/m4])
AC_CONFIG_FILES([tests/Makefile tests/atlocal])
```

Given that the current version of Automake, 1.5, does not provide Autotest support, one merely uses the regular Makefile snippets in '`Makefile.am`' ((**FIXME:** *Embedding an Autotest Test Suite.*), and in particular *example 12.5*).

One interesting bit is the handling of the generated tests. Because the source files of the tests are expected to be found in the source tree, even though '`generated.at`' is generated (surprise!), we have to qualify its path:

```
m4_texinfo = $(top_srcdir)/doc/m4.texinfo
generate   = $(AWK) -f $(srcdir)/generate.awk
$(srcdir)/generated.at: $(srcdir)/generate.awk $(m4_texinfo)
        $(generate) $(m4_texinfo) >$@t
        mv $@t $@
```

As already described in (**FIXME:** *Embedding an Autotest Test Suite.*), you hook the `testsuite` to Automake's `check-local` target:

```
check-local: atconfig $(TESTSUITE)
        $(SHELL) $(srcdir)/$(TESTSUITE)
```

and finally, after so many pages to read, you can run happily '`make check`', and stare happily at

```
## ---------------------------- ##
## All 76 tests were successful. ##
## ---------------------------- ##
```

One of the most important targets provided by Automake is `distcheck`, which basically checks that your packages behaves properly: it compiles cleanly when using a separate build directory, the test suite succeeds, installs the package in some temporary directory etc. This gives you the guarantee that all the files needed to compile and test your package are shipped. Run it, and see how Autotest boxes are so much more beautiful than Automake's...

```
...
## ---------------------------- ##
## All 76 tests were successful. ##
## ---------------------------- ##
...
========================================
m4-1.5.tar.gz is ready for distribution
========================================
```

Unfortunately, although a plain `distcheck` is a very significant sign that your package behaves properly, it offers no guarantee that you did not forget to install some files! Don't laugh, I have already been trapped; it even happened that I produced incorrect paths in installed configuration files, while the test suite and `distcheck` were very happy because... they were *not* using the package as a user would do, they were bypassing configuration files etc. Lack of realism...

Fortunately Automake provides the `installcheck` target, which is run by `distcheck`, *after the package was installed*! Alleluia! Hook the test suite to `installcheck`, but setting the `AUTOTEST_PATH` so that the *installed* `m4` be run:

```
# Run the test suite on the *installed* tree.
installcheck-local:
        $(SHELL) $(TESTSUITE) AUTOTEST_PATH=$(exec_prefix)/bin
```

This time, we really did all we could to test our package.

13 Source Code Management with CVS

Now that we have built up a huge example project, filled with source code, test code and portability scripts, it is time to protect it and organize it. We want to protect it from accidental deletion, and we want to organize the archiving of it so that we can find particular revisions of the software very easily. These activities fall under the rubric of "Configuration Management".

13.1 Why the bother

Configuration Management is crucial to building reliable systems consistently. It encompases not only software compatiblities, but hardware, infrastructure and many other issues. This is a software book.

Software Configuration Management is crucial to building reliable software systems consistently. It encompasses source code management, tracking status, correlating changes with problem reports, recording versions of tools used to build products and many other issues important for auditing past product builds and repeatability in building past and current products. The bibleography will help you find resources for these advanced topics. This book is focused on developing for relatively smaller projects without comprehensive repeatability requirements. Here, we cover Source Code Management.

On any development project, especially one involving several people, a certain amount of confusion is inevitable. Writing solid code quickly is a matter of organizing the chaos. Configuration management in general and source code management in particular is basically for the purpose of managing (taking control of) the chaos.

The most important elements in this are

- Keeping track of the history of the evolution of the software.
- Allowing you to extract copies of the software at marked and unmarked points in time. Most likely, points in time that correspond to releases.
- Providing a branching capability so that you can stabilize a release branch while continuing the development on the main code branch.
- Providing a branching capability so that you can **de**stabilize a branch on a special project involving substantial rework.
- Supporting the re-integration of work that was done on one branch into another branch. For example, in the interest of reducing chaos you do not want all the reintegration of a long project to occur exclusively at the end of the project. That way lies madness.

  With the advent of graphical diff/merge tools, great strides have been made here in the last few years.

There are many open source tools available that meet these criteria, See Section 13.9 [Other Resources], page 246. However, we will focus on CVS, since it is more widely used and understood than the others. So, we will give you just enough information to be dangerous.

Even still, there is other useful, but not as crucial, criteria in choosing one product over another:

- Does it support configuration management capabilities? Viz.,

  '`change management`'
  : It can be very useful to be able to track changes related to particular bug reports and development projects.

  '`build support`'
  : Many SCM products are integrated with certain or various build tools. These products ensure that the software build process produces results based on consistent versions of the source. It also eliminates the situation where only build

meisters have the knowledge and skills to build a product. Such situations are common and not generally a "good thing."

'`release support`'
: A few SCM products actually track which customers have which versions of which products. Using that information, it is possible to focus "emergency releases" on only the affected customers. Doing that is a good thing.

'`process management`'
: It is useful to understand how you do what you do, especially if you are trying to meet ISO 9000 or capability maturity model requirements. It is very easy for such procedures to become enmeshed in otherwise useless bureaucratic paperwork. Some of the tools can facilitate some of this burden. Programmers prefer that.

- Is the interface integrated and consistent? You do not want to have to refer to documentation all the time. The functions and interfaces should be generally intuitive and obvious, based on simple introductory information.
- Is the functionality accessable with a GUI interface? Good GUI's can make doing things much more obvious.
- over a network Since individual workstations tend to not be backed up, you definitely want to try to centralize your repositories on systems that *do* get backed up.

These are "things to consider" for large scale institutional SCM implementations.

13.2 Creating a new CVS repository

In general, it is better to have a single repository that can be professionally maintained and backed up. That may not be possible if there is none available or if there are reasons why you must keep your work in a separate repository. In such cases, you will need to create your own. Fortunately, that is easily done.

First, you must determine where you can and should place it. The repository will start out somewhat larger than the original sources and will tend to grow monotonically with time. That is to say, it is rare to see the size do anything except get bigger. Much bigger. Another reason for looking beyond your home directory is that you may wish to share the data with others. Home directories are considered fairly private places.

Before we go and create the repository, though, a few things need to be decided. If this new repository is for the use of a single user, you need to make sure that the created repository has user-only write permissions on the directories. Otherwise, access control in CVS is generally based on group membership permissions. Consequently, it is generally a good idea to set up a special purpose group ID so that the various CVS users can become members of it.

Now, create the repository directory with the appropriate group ownership and group write privileges:

```
CVSROOT=/path/to/repository
cvs -d $CVSROOT init
find $CVSROOT | xargs chmod g+w
```

For a personal repository, the only required command is the "'`cvs init`'" command. For a shared repository, the "'`find ... | xargs chmod`'" thing is crucial in order to allow other group members access to the repository.

At this point, it would also be convenient to save the value of the CVSROOT environment variable where your login shell will find it for the next time you login. You can do this by editing your initialization script file for your particular shell to contain either this:

```
CVSROOT=/path/to/repository
export CVSROOT
```

or something fairly similar, if you use '`csh`' or a derivative. CVS uses that environment variable when it cannot locate "'`./CVS/Root`'".

At this point, you now have a working CVS repository. It's just empty. It is time to either start a new project, See Section 13.3 [Starting a new project], page 245, or install a pre-existing project, See Section 13.4 [Installing a pre-existing project], page 245.

13.3 Starting a new project

If your project is a component of a larger project, you would extract ('`checkout`') a copy of the source tree, '`cd`' into the appropriate directory and:

```
mkdir proj-dir-name
cvs add proj-dir-name
cd proj-dir-name
```

and then create and '`cvs add`' the new files and directories that constitute the new project.

If the project is to be completely separate from other projects in the repository, then you must create some of the files and directories that will constitute the new project and '`import`' them as if you were installing a pre-existing project, See Section 13.4 [Installing a pre-existing project], page 245.

13.4 Installing a pre-existing project

To install a pre-existing project into a CVS repository, you '`cvs import`' the source tree.

13.5 Extracting a copy of the source

13.6 Returning changes to the repository

13.7 Marking the revisions in a release

13.8 Bibliography

[FOGEL99] _Open Source Development with CVS_ Karl Fogel; Coriolis Open Press, Scottsdale, Arizona, 1999

This book was written by a co-founder of Cyclic Software, the company that underwrites and manages most of the CVS development. This is a very comprehensive treatment of CVS. Useful if you intend to manage a repository for others' use.

[BABICH86] _Software Configuration Management_ by Wayne A. Babich; Addison-Wesley, Reading, Massachusetts, 1986

[BERLACK92] _Software Configuration Management_ by H. Ronald Berlack; John Wiley and Sons, Inc., New York, New York, 1992

[BUCKLEY92] _Implementing Configuration Management: Hardware, Software and Firmware_ by Fletcher J. Buckley; IEEE Press, 1992.

[COMPTON93] _Configuration Management for Software_ by Stephen B. Compton and Guy R. Conner;Van Nostrand Reinhold; John Wiley and Sons, Inc., New York, New York, 1993

13.9 Other Resources

‘CVS’ http://www.cvshome.org

‘BitKeeper’
http://www.bitmover.com/bitkeeper

‘PRCS’ http://prcs.sourceforge.net

‘aegis’ http://www.canb.auug.org.au/~millerp/aegis/aegis.html

‘subversion’
http://subversion.tigris.org/

‘General Information’
http://www.cmtoday.com/yp/configuration_management.html

‘news group’
news://comp.software.config-mgmt

14 Debugging with gdb and DDD

This chapter will explain what debugging is, and the tools available to carry it out. There are debugging tools which allow you to watch a program as it runs, and even change what it does to a small extent. There are others which replace library routines such as malloc with other rutines which give far more information and allow you to track down problems such as memory which is allocated but no longer used (memory leaks), or pointers which point to places that aren't valid anymore.

14.1 Why Do I Want A Debugger?

When you have finally got your program to compile with no errors, the next step is to execute it. If you are incredibly lucky or an astonishingly good programmer, then you can skip this chapter. Otherwise you will sooner or later need to use a debugger.

Once the program is executing, it may do any of a number of things. Typically, it will do what you expected, except for a few small details - taking the wrong branch, or calculating values which are too high or too low, or just plain ridiculous.

Most of these problems are head-slappers - you understand immediately what is wrong and run through the edit-compile-test cycle again. An example of this would be FIXME

The next stage you get to is the more subtle problems. This is where the program runs for a long time then inexplicably dies with a core dump, or one of the ouput values occasionally comes out negative when it shouldn't be able to. That's when you need a debugger.

A debugger allows you to take control of the running program. You can start it, stop it, make it run one line at a time, examine the values of variables at each step, and even change their values to see what would happen. Sometimes it may take 20 minutes for a program to run to the point where it all goes wrong - you can put a breakpoint in the code and the debugger will allow it to run normally until that point, then stop and give you full control.

14.2 How to Use a Debugger

14.2.1 A Program With Bugs

This is the example program we'll be using to demonstrate how you would typically use gdb to track down bugs. It has a number of bugs, some obvious, some not so obvious. It compiles with no warnings using 'gcc -g ecount1.c -o ecount1'

The program has a simple task to perform - it takes one parameter, a single word, and counts the number of letter 'e's in it. It prints the result to the screen. That shouldn't be too difficult, should it?

```
/*
 * ecount1.c Simple program to count the number of letter 'e's that
 * appear in the word given as the only parameter
 *
 * WARNING: This program contains several deliberate bugs
 *  (and possibly some others...)
 */

#include <stdio.h>
```

```
int main( int argc, char *argv[] )
{
  char *buf=NULL;
  int count, i;

  /* check we have a parameter */

  if( argc = 1 )
  {
    printf( "Usage: ecount <word>\n");
    exit(1);
  }

  /* Make our own local copy of argv[1] */

  strcpy( buf, argv[1] );

  /* print it out to show we received it correctly

  printf( "The word is '%s'\n", buf );

  /* Now step through counting letter 'e's */

  for( i=0; i<strlen(buf); ++i )
  {
    if( buf[i] == 'e' ) ++count;
  }

  return(0);
}
```

14.2.2 Compiler Options

While it is possible to run any executable under a debugger, it is much easier if you use some compiler options and don't use others. Usually, to make the executable smaller, much of the symbolic information (variable names, function names etc.) are thrown away by the compiler when it has finished with them. This means that the debugger can't tell what the value it's looking at is called, and makes the debugging session very cryptic.

The way around this is to use the '-g' compiler flag. This causes the compiler to insert lots of debugging information into the executable, so the debugger knows everything it needs to know. Using this flag will make the executable noticeably bigger, but won't affect the performance too much - it just appends tables of symbolic information to the executable file.

A lot of people are under the impression that the '-g' debugging flag makes a program run slower. The reason for this is that to use a debugger, you generally need to turn off any optimisation flags ('-O' or '-O1-9'). This can make a significant difference to the speed of the program, as the optimisers on modern compilers are pretty good.

The GNU compiler collection, gcc, is unusual in that it allows you to use '-g' alongside '-O' flags. While this seems like a good thing, there are hidden dangers - the optimiser may have rearranged the order of some bits of your code, and variables you used may not be changed when you think they should be, or they may have disappeared altogether!

In general, then, always use *either* '-g' *or* '-O', not both together.

14.2.3 The First Attempt

To start with, we need to compile our example. Following the above advice about compiler flags, we will use the following command line to compile it:

'`gcc -g ecount1.c -o ecount1`'

This will make the executable '`ecount1`' from the source '`ecount1.c`', using the -g flag to turn on debugging information in the executable and the -o flag to name the output file '`ecount1`' instead of the default '`a.out`'.

Now we will try to run the program and see what happens.

'`bash$ ecount1`' '`Usage: ecount <word>`'

We forgot to include the word for it to count the number of 'e's in! So it correctly gave us a usage message. Let's try again.

'`bash$ ecount1 example`' '`Usage: ecount <word>`'

Oh dear. Even when we include an argument, it still gives us an error message. We could probably fix this one by inspecting the code, but this chapter is about using the debugger, so let's do that...

14.3 An example debugging session using gdb

The program was compiled with the '-g' flag set, so you can run it under control of the debugger. The easiest way to do this is to start the debugger with the name of the program as an argument. I will be describing 'gdb', the GNU debugger, so for the example program you would use '`gdb ecount1`'.

This makes the debugger start up, load the executable and all of it's source code and variable names into memory, then wait for your command. At this point the program is not yet running, but the debugger knows everything it needs to know about it.

```
bash$ gdb ecount1
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"...
(gdb)
```

'`gdb`' is now ready to run the program.

14.3.1 Attaching to an Already Running Program

It is possible that the program is not started directly from the command line - it might be run from another program, or be created by an incoming network request.

In this case it may not be possible to run it independently and still observe its behaviour. All is not lost, however. If you have the process ID of the process you want to debug, then you can just use '`gdb executable-name process-id`' and gdb will attach itself to the running process and stop it wherever it is, ready for your commands.

Alternatively you can start gdb as if you were about to debug the executable normally, with '`gdb executable-name`', then type '`attach`' at the command line.

This will give you a menu of all the processes of that name on the system so you can select the one you want FIXME

Tip: if the process starts up then immediately crashes, then you can add a line such as '`sleep(30);`' immediately after the start of '`main()`' which will give you 30 seconds to get the PID and attach to the process. Remember to take this '`sleep()`' out again before releasing the code. DAMHIKT. FIXME

14.3.2 Running the Executable

Since we are now at the gdb prompt with our program loaded, let's start it. For a program with no arguments, you can just type '`run`', but for our program we need a word for it to work on, so we use '`run example`'. Note that gdb already knows the executable name, so you just give the '`run`' command the argument(s).

If you need to run a program many times in a debugging session, then it remembers the arguments you used last time, and passes them in again unless you specify others. The way to run it with no arguments again is to use '`set args`'. You can use '`show args`' to see what arguments will be used next time you type '`run`'.

So, let's run it:

```
(gdb) run example
Starting program: /home/paul/gnuprog2/src/debugging/ecount1 example

Program exited with code 01.
(gdb)
```

First problem - the program just ran right through to the end without stopping and finished.

So we need a way to stop it and take control.

14.3.3 Taking Control of the Running Program - Breakpoints

Breakpoints are special markers the debugger uses to tell it where to stop. When a breakpoint is set on a particular line of code, then the program is allowed to run at normal speed until it reaches that line, then it is stopped and the debugger takes over. It puts up the prompt and allows you to control the program however you like.

Since this program is so short, we will place a breakpoint on the '`main()`' function, so we take control as soon as the program starts.

'`gdb`' understands a number of ways to specify where to place a breakpoint, such as a line number, or a function name, or a certain number of lines forwards or backwards from the current position. We want to stop on entry to '`main`' so we'll use '`break main`'.

```
(gdb) break main
Breakpoint 1 at 0x8048496: file ecount1.c, line 13.
(gdb) run example
Starting program: /home/paul/gnuprog2/src/debugging/ecount1 example

Breakpoint 1, main (argc=2, argv=0xbffff914) at ecount1.c:13
13   char *buf=NULL;
(gdb)
```

'`gdb`' is now stopped at the beginning of main waiting for us to tell it what to do.Notice that the breakpoint was actually set at line 13, which is the first line of actual code in the program. '`gdb`' cannot set a breakpoint on lines where nothing is actually executed (comments, function declarations etc.). If you ask it to, then it will place it at the next available location.

14.3.4 One Step at a Time - Step+Next

What we need to do now is run the program a line at a time and see if we can spot the problem that is causing us to get a '`Usage:`' message even when the arguments are correct.

Let's look at the program again. We'll use the `gdb` '`list`' command to examine the code.

```
(gdb) list
8
9 #include <stdio.h>
10
11 int main( int argc, char *argv[] )
12 {
13   char *buf=NULL;
14   int count, i;
15
16   /* check we have a parameter */
17
18   if( argc = 1 )
```

'`gdb`' is sat at line 13, where NULL is assigned to '`buf`'. There are two commands which will cause us to move one line forward in the file, '`step`' and '`next`'.

'`step`' will always go to the next line to be executed, following function calls if there are any on the line. '`next`' will run until the next line which executes without following function calls. A large amount of your program can be run between two '`next`' commands, whereas only one line of code can run between '`step`' commands.

Since we have no functions to go into, we'll generally use '`step`'.

```
(gdb) next
18   if( argc = 1 )
(gdb)
```

We are now on line 18. Why didn't it run to line 14? That's because there are only variable declarations on that line, so no code was actually executed for that line. All it does is tell the compiler those variables exist.

14.3.5 Examining Variables - Print

So now we are at an '`if`' statement. This is the one that is going wrong, so let's examine it. The only thing we can check is '`argc`', so let's do that.

```
(gdb) print argc
$1 = 2
(gdb)
```

This is slightly cryptic, so let's look a bit deeper at it. The answer is 2, which is what we expect for argc with one argument given (one element for the name of the program, one for the argument).

The '`$1`' is the identifier given this value in '`gdb`'s 'value history'. This works like the history in a shell, storing the values you have looked at ready for you to refer back to later.

FIXME

For instance, if you '`print`'ed a complicated expression such as '`myptr+i*sizeof(struct mystruct)`' to get the result '`$27 = 0xffffeb2c`', then to see the contents of that structure you could use '`print *$27`' instead of repeating the calculation or copying the hex address.

Use '`show values`' to see the last ten values in the history, or '`show values n`' to show the ten values centred on history item number '`n`'.

14.3.6 The First Bug

The value of '`argc`' was correct, so lets move on a step and see what happens.

```
(gdb) next
20     printf( "Usage: ecount <word>\n");
(gdb) list
15
16   /* check we have a parameter */
17
18   if( argc = 1 )
19   {
20     printf( "Usage: ecount <word>\n");
21     exit(1);
22   }
23
24   /* Make our own local copy of argv[1] */
(gdb) print argc
$2 = 1
(gdb)
```

Here we see that execution moved into the '`if()`' statement, which we didn't expect based on the value of '`argc`'. I have listed the code to remind myself of the surrounding structure, then checked the value of '`argc`' again.

'`argc`' is now 1, not 2! Looking closer at the code we see that the comparison operator in the '`if()`' statement was incorrectly written as '`=`' instead of '`==`'. This assigned 1 to '`argc`' and returned 1 as the expression value, so the '`if`' evaluated as true and execution went into the '`if`' block.

14.3.7 Try Again...

Let's correct this and try again. For example purposes, the new code is in ecount2.c.

```
bash$ gcc -g ecount2.c -o ecount2
bash$ ecount2 example
Segmentation fault (core dumped)
```

14.3.8 Core Dumps - What Are They?

Now the program has caused a '`Segmentation fault`' and dumped a '`core`'.

(If your program just said '`Segmentation fault`' and didn't say '`core dumped`' then your shell has core dumps disabled. Type '`ulimit -c 10000000`' to enable them and run ecount2 again)

What is a '`Segmentation fault`'? How about a '`core dump`'?

Part of the multi-user aspect of UNIX is that programs that attempt to write to memory that doesn't belong to them get caught and killed by the operating system. This is a major factor in the long term stability of UNIX compared to lesser OSes.

A '`segmentation fault`' happens when a process attempts to write to memory outside of the address range that the OS has tagged as accessible to it. Since this is a good sign that the process is out of control, the OS takes immediate and severe action, and kills the process.

To aid in debugging the process, the OS then places a copy of all the memory occupied by the program and it's data into a '`core`' file, which it stores in the programs '`current directory`'.

Since in our example we haven't changed that directory, there should now be a file called '`core`' in the same place as '`ecount2.c`' and '`ecount2`'.

```
bash$ ls -l
total 320
-rw-------    1 paul     users       65536 Sep  5 21:11 core
-rwxr-xr-x    1 paul     users       21934 Sep  5 20:45 ecount1
-rw-r--r--    1 paul     users         732 Aug 30 21:01 ecount1.c
-rwxr-xr-x    1 paul     users       22006 Sep  5 20:45 ecount2
-rw-r--r--    1 paul     users         790 Aug 30 21:02 ecount2.c
```

Here we see that the file called '`core`' is present in the working directory, and has been given permissions for the user *only* to read and write it. This is because the core file will contain all the information the program knew at the time it crashed, which may include passwords or personal information you had entered into it.

If your program uses lots of memory, then the core dump will be at least as big as the executable size plus the total data size, and possibly a lot bigger, as the OS will dump every '`page`' of memory the program has used in its entirety. This is the reason why some default configurations disable core dumps - they are only useful if you are in a position to debug the process, and can be several megabytes in size.

14.3.9 How to Use a Core Dump

To use the core dump, we give it as another argument to '`gdb`':

```
bash$ gdb ecount2 core
Core was generated by `ecount2 example'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  strcpy (dest=0x0, src=0xbffffa9d "example")
    at ../sysdeps/generic/strcpy.c:40
40 ../sysdeps/generic/strcpy.c: No such file or directory.
(gdb)
```

Here we see that '`gdb`' loads the executable as usual, but also loads the core file. It can tell us what arguments the program was called with, and why the core was dumped (Segmentation Fault in this case).

'`gdb`' then loads the symbol tables of all the shared libraries that our program had loaded at the time it crashed, so it has all the information at its fingertips.

Next we have a line starting with '`#0`' which tells us that the program crashed in routine '`strcpy()`' with two arguments, '`dest=0x0`' and '`src=0xbffffa9d`', which '`gdb`' helpfully expands to show that it points to the string '`example`'.

This function is in one of the system libraries, so although '`gdb`' knows which source file and line the crash occurred on, it has not got access to the source and complains. Thanks to the wonders of Open Source, we could get the source code for the appropriate library and tell '`gdb`' via its '`directory`' command. Try '`help directory`' in '`gdb`' for more details.

Usually, you don't really need to get into library source, unless you know you are using a locally produced or bleeding edge library, so we won't worry about that.

If we look up the manpage for '`strcpy()`', we find that it takes two parameters, the first a char * pointing to the space to store the copy, and secondly the string to be copied from.

In the line starting #0 we see that the first parameter is a NULL pointer. Trying to write to this is what caused the segmentation fault which killed the program.

That's all well and good, but where in our code is this happening?

14.3.10 Finding Out Where You Are - Backtrace

'gdb' keeps track of where you are in your program and how you got there. The line starting with '#0' is the first entry in the 'call stack' which it keeps for all this information. If you use the command 'backtrace' (for which you can use the abbreviation 'bt' or the synonym 'where'), then it will show you the whole call stack.

```
(gdb) backtrace
#0  strcpy (dest=0x0, src=0xbffffa9d "example")
    at ../sysdeps/generic/strcpy.c:40
#1  0x80484d5 in main (argc=2, argv=0xbffff964) at ecount2.c:26
#2  0x400382eb in __libc_start_main (main=0x8048490 <main>, argc=2,
    ubp_av=0xbffff964, init=0x8048308 <_init>, fini=0x804856c <_fini>,
    rtld_fini=0x4000c130 <_dl_fini>, stack_end=0xbffff95c)
    at ../sysdeps/generic/libc-start.c:129
(gdb)
```

Here we see that the call to 'strcpy()' came from line 26 of 'main()'. Above that in the call stack (although confusingly 'gdb' prints the stack out lowest level first...) is the '__libc_start_main()' function, which is used by the compiler to do any initialisation etc. that it needs to before calling our 'main()' function. It is hardly ever worth going any higher than main unless you are debugging compilers themselves...

14.3.11 Moving Around the Call Stack - Up+Down

We now have just enough information to find the line which caused the segmentation fault, but it would be nice to have 'gdb' sat on that line so we can use the source listing and variable prnting commands to find out why it happened.

'gdb' will only allow you to examine variables which are in scope at the place where it is, so we cannot examine local variables of 'main()' from inside 'strcpy'.

The answer to this is the 'up' and 'down' commands. As the names suggest, 'up' moves one step up te call stack, to '#1 main()', while 'down' moves down one step. We can't go down from here as we are already at the lowest level, so let's try 'up'

```
(gdb) up
#1  0x80484d5 in main (argc=2, argv=0xbffff964) at ecount2.c:26
26   strcpy( buf, argv[1] );
(gdb) list
21     exit(1);
22   }
23
24   /* Make our own local copy of argv[1] */
25
26   strcpy( buf, argv[1] );
27
28   /* print it out to show we received it correctly
29
30   printf( "The word is '%s'\n", buf );
```

Now we are back in the situation we were in before, at line 26 in main.c. We can examine all the variables and list the source. Note that we can't use the '`step`' and '`next`' commands from here because we don't actually have a running program to step through, only an image of the state it was in when it crashed.

We can now see that the first argument to '`strcpy`' was the variable '`buf`', so let's look at what has been done to that. In a comlex program, we would use an editor or even a class browser to do this, but here we know it is defined at the top of main, so let's have a look there.

```
(gdb) list main
8
9 #include <stdio.h>
10
11 int main( int argc, char *argv[] )
12 {
13   char *buf=NULL;
14   int count, i;
15
16   /* check we have a parameter */
17
(gdb)
```

So '`buf`' was initialised to NULL, and no memory was ever allocated to it. I think we have found our bug. Let's fix it and try again with '`ecount3.c`'.

14.3.12 The Third Bug

Now we have changed '`char *buf;`' to '`char buf[10];`'. This will do for our example, but in 'real' code we should really use a safer mechanism such as using '`malloc`' to allocate the right amount of space (including the terminating '`'\0'`', then using the safer '`strncpy()`' to copy the string into it.

Let's run '`ecount3`' and see what happens.

```
bash$ ecount3 example
There are 1074045242 'e's in 'example'.
```

Can you spot what went wrong there? I only count 2 'e's in 'example'.

Lets fire up '`gdb`' again and see where this number comes from.

14.3.13 Fun With Uninitialised Variables

```
bash$ gdb ecount3
(gdb) b main
Breakpoint 1 at 0x8048496: file ecount3.c, line 18.
(gdb) r example
Starting program: /home/paul/gnuprog2/src/debugging/ecount3 example

Breakpoint 1, main (argc=2, argv=0xbffff914) at ecount3.c:18
18   if( argc == 1 )
(gdb) n
26   strcpy( buf, argv[1] );
```

Now we are on the line that first uses the string we supplied. We could test it here, but '`gdb`' prints the line we are *about to execute*, not the line just done. So '`buf`' won't yet have the string in it. Let's move on one more line and see what's in '`buf`' then.

```
(gdb) n
34   for( i=0; i<strlen(buf); ++i )
(gdb) p buf
$1 = "example\000"
```

So ‘buf’ does have the right string in it - but ‘gdb’ has printed the whole of the char array, since it knows how long it is. This leads to it showing the ‘’\0’’ on the end, plus some trailing garbage. Usually we could just ignore that, but for the purposes of the example I’ll tidy it up.

```
(gdb) set print null-stop
(gdb) p buf
$4 = "example"
```

‘set print null-stop’ tells ‘gdb’ to treat char arrays like C strings, and stop when it reaches a ‘’\0’’.

Now let’s get back to this crazy value for ‘count’. We’ll start off by printing it’s value before we’ve done anything to it.

```
(gdb) p count
$6 = 1074045240
```

Aha! It starts off with a very high value - we forgot to initialise it in the first place. Rather than go through the whole edit-recompile-build cycle again, let’s use a powerful feature of ‘gdb’ and edit it in place.

```
(gdb) set count=0
(gdb) p count
$7 = 0
```

So now it’s initialised as it should be (and will be once we’ve put it right....if we remember!).

We are at the start of the loop that counts ’e’s, so let’s step inside it then have a good look at what goes on during the loop. Here we *could* use ‘print’ after every step, but if we use ‘display’ instead, then ‘gdb’ will automatically print the values every time it stops.

```
(gdb) n
36      if( buf[i] == 'e' ) ++count;
(gdb) display count
1: count = 0
(gdb) display buf[i]
2: buf[i] = 101 'e'
```

Now we can step through the loop and watch the value of ‘count’ change along with the letter it’s currently examining.

```
(gdb) n
34   for( i=0; i<strlen(buf); ++i )
2: buf[i] = 101 'e'
1: count = 1
(gdb) n
36     if( buf[i] == 'e' ) ++count;
2: buf[i] = 120 'x'
1: count = 1
(gdb) n
34   for( i=0; i<strlen(buf); ++i )
2: buf[i] = 120 'x'
1: count = 1
(gdb) n
36     if( buf[i] == 'e' ) ++count;
2: buf[i] = 97 'a'
```

```
1: count = 1
(gdb) n
34   for( i=0; i<strlen(buf); ++i )
2: buf[i] = 97 'a'
1: count = 1
(gdb) n
36     if( buf[i] == 'e' ) ++count;
2: buf[i] = 109 'm'
1: count = 1
(gdb) n
34   for( i=0; i<strlen(buf); ++i )
2: buf[i] = 109 'm'
1: count = 1
(gdb) n
36     if( buf[i] == 'e' ) ++count;
2: buf[i] = 112 'p'
1: count = 1
(gdb) n
34   for( i=0; i<strlen(buf); ++i )
2: buf[i] = 112 'p'
1: count = 1
(gdb) n
36     if( buf[i] == 'e' ) ++count;
2: buf[i] = 108 'l'
1: count = 1
(gdb) n
34   for( i=0; i<strlen(buf); ++i )
2: buf[i] = 108 'l'
1: count = 1
(gdb) n
36     if( buf[i] == 'e' ) ++count;
2: buf[i] = 101 'e'
1: count = 1
(gdb) n
34   for( i=0; i<strlen(buf); ++i )
2: buf[i] = 101 'e'
1: count = 2
(gdb) n
39   printf( "There are %d 'e's in '%s'.\n", count, buf );
2: buf[i] = 0 '\000'
1: count = 2
```

So now we've stepped through the loop and found the two 'e's correctly. The two '`display`' expressions are still active, so we'd better stop them.

```
(gdb) undisplay
Delete all auto-display expressions? (y or n) y
(gdb) n
There are 2 'e's in 'example'.
```

All looks good so far, once we've made the change to the initialisation of '`count`'.

14.3.14 Try Again - Again...

Now we have example program '`ecount4.c`' which looks like this at the top of '`main()`'

```
11 int main( int argc, char *argv[] )
12 {
13   char buf[10];
14   int count=0, i;
```

We compile it with '`gcc -g ecount4.c -o ecount4`' and get the following result:

```
bash$ ecount4 example
There are 2 'e's in 'example'.
```

14.3.15 Success! ...or is it?

So we can be proud of ourselves now - we have a working program at last!

Hold on a moment, though - let's look back through the code.

```
24   /* Make our own local copy of argv[1] */
25
26   strcpy( buf, argv[1] );
27
28   /* print it out to show we received it correctly
29
30   printf( "The word is '%s'\n", buf );
31
32   /* Now step through counting letter 'e's */
33
34   for( i=0; i<strlen(buf); ++i )
```

What happened to line 30? We should have seen the word printed out, but there was no sign of it.

Looks like we have yet another bug to find. (I'm glad I made this program up to demonstrate bug finding - I'd hate to make this many errors in a real program. Although I have had days like that...)

14.3.16 The Fourth Bug

Let's fire up '`gdb`' and see what's going on.

```
bash$ gdb ecount4
(gdb) list 30
25
26   strcpy( buf, argv[1] );
27
28   /* print it out to show we received it correctly
29
30   printf( "The word is '%s'\n", buf );
31
32   /* Now step through counting letter 'e's */
33
34   for( i=0; i<strlen(buf); ++i )
(gdb) b 30
```

We've started 'gdb', looked at the lines around line 30 and seen nothing obviously wrong. Notice we have given the list command a line number to work with - it will display lines centred on the supplied line number.

Then we've set a breakpoint on line 30, using 'b 30'. Many 'gdb' commands can be abbreviated to their shortest unique abbreviation.

The next step is to run the program. It should stop on the line which seems not to be being executed.

```
Breakpoint 1 at 0x80484d8: file ecount4.c, line 30.
(gdb) run example
Starting program: /home/paul/gnuprog2/src/debugging/ecount4 example

Breakpoint 1, main (argc=2, argv=0xbffff914) at ecount4.c:34
34   for( i=0; i<strlen(buf); ++i )
```

Well it *should* have stopped on line 30, but it has actually stopped at line 34, at the start of the counting loop. Let's stop the program in mid-execution and put a breakpoint earlier, so we can see what happens.

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) b 26
Breakpoint 2 at 0x80484c0: file ecount4.c, line 26.
(gdb) run
Starting program: /home/paul/gnuprog2/src/debugging/ecount4 example

Breakpoint 2, main (argc=2, argv=0xbffff914) at ecount4.c:26
26   strcpy( buf, argv[1] );
(gdb) n

Breakpoint 1, main (argc=2, argv=0xbffff914) at ecount4.c:34
34   for( i=0; i<strlen(buf); ++i )
```

'kill' will stop the program being run, taking it back to the state it was in when 'gdb' first started up. All breakpoints, displays etc. will remain set up, so this is useful for cases like this where we want to keep going over the same part of a program without exiting and re-entering 'gdb' and setting up all the breakpoints again.

When we run it again, we see that it definitely goes from line 26 ('strcpy()') to line 34 ('for() loop'). There is no doubt in 'gdb''s mind that line 30 is just not there. The only way this could happen is if it was removed by the preprocessor, either by a macro substitution or a comment.

Closer examination reveals that the comment on line 28 is not closed - so the preprocessor will have just carried on reading and throwing lines of code away until it came to a comment closing sequence - ''*/''. This is on line 32, which explains why line 30 was being missed out.

14.3.17 One More Time...

When we recompile as 'ecount5' we get this result:

```
bash$ ecount5 example
The word is 'example'
There are 2 'e's in 'example'.
```

Hurray! Finally it has worked correctly.

There are still a few things in the program which *will* catch us out later, though.

For instance, when we made buf into a local array defined on the stack, we only made it ten characters long. What will happen if someone enters '`Supercalifragilisticexpialidocious`' as a word? It will scribble all over the stack and the results will be completely undefined - it may work, it may crash, or most dangerous of all it may appear to work but actually have corrupted some other variables or the return address of the function, so it will crash when it returns, much later on...

14.3.18 So What Have We Learned?

This is the end of the first section of this chapter. We have gone through a simple program with lots of bugs, and seen how to use a debugger to investigate the problem.

We can run the debugger with '`gdb programname`', then run it with parameters using '`run param1 param2`'. Breakpoints are inserted with '`break main`' or '`break 34`' to force the program to stop at the start of '`main()`' or at line 34 respectively.

Once the program has stopped and is under our control, we made it go one step at a time using '`step`' or '`next`'. At each step we could examine the value of local or global variables which were in scope at that point using '`print`'.

We found out what a '`core dump`' actually is, and how to examine it to see what caused the program to crash, and where, using '`gdb programname core`'. '`backtrace`' and '`up`' and '`down`' are used to move around the stack frames to see what led us to where the program crashed.

We saw how '`gdb`' has the power to alter variables while the program is running, to try out potential solutions without having to recompile the whole thing.

Now you know enough to carry out most of the day-to-day debugging most people need to do.

There are two parts to the rest of the chapter - the next one will look at GUI front ends which may make your debugging task easier, then the final one looks at more complicated debugging scenarios - debugging multithreaded code, overloaded C++ functions, making it stop only when it's just about to go wrong, that sort of thing.

14.4 Debugging the Pretty Way - GUI Front ends

14.4.1 Why a GUI?

14.4.2 Whatś the choice?

14.4.3 DDD - Some History

14.4.4 Revisiting the same example

(DDD - the easy way (now we've seen the 'hard' way). Give a brief description, use lots of screenshots since it's main advantage is the GUI.)

(Run through doing the same example as with gdb, but in a friendly' way. And quicker.)

(Show off the extra abilities you have with DDD: point and click breakpoints, Dynamic Data Display, array graphing using gnuplot etc.)

14.5 More complicated debugging with 'gdb'

(Complicated programs - multithreaded, multiple processes)

(Examining variables and memory in detail - looking at the stack, registers, symbol table info - also dumping data in hex, arbitrary formatting, print options, machine code)

(Making it go wrong - sending signals, jumping to the faulty code, forcing variables)

(Stopping in the right place - conditional breakpoints, watchpoints, catchpoints)

(Better use of the command line - readline commands, macros, completion)

(finding the source code, floating point hardware support, type/range checking)

(Using 'gdb' internal variables to help debugging)

(Other language support - C++ specific, Java, others)

(The future of debugging (e.g. The next major release of DDD understands STL containers))

14.6 Other debugging aids

(Mention some of the other free debugging tools: dmalloc, mprof, strace, fuser mpatrol etc.)

14.6.1 Memory checkers

14.6.2 Debugging uncooperative programs

(strace)

15 Profiling and Optimising Your Code

* CHAPTER 14: Profiling and Optimising Your Code * c. 15kwords by Paul Scott

WTF is profiling? Explains what it is and why you'd want to do it. (Case study: I had a program that logged ~2M events/day taking lots of CPU. Profiling showed mktime() using 96.4% of the time as it was called for each one. I changed it to call mktime() once for the first report of each day and calculate the offset; the program now only takes 4% of original load.)

How to profile your program - compiler options. Also go into profileable system libraries if needed.

What to do when running a program to get the most useful information from the profiling

** Generating the profile by running gprof

Understanding the output - nested call graphs, 'spontaneous' functions, what the percentage figure means as opposed to system+user or wall time.

Modifying your code to get better profiling information - e.g. splitting up large, slow functions into a chain of smaller ones to zoom in on the hotspots (only for use while profiling).

When to try and fix it and when not to. Also when to stop.

** Optimising - what it actually means. Go into the O(n) notation to give a rough idea.

Talk about the order in which to do things: 1. Get it working 2. Get it working properly ;-) 3. Figure out how much effort is worthwhile: will it be run once a year, or twenty times every day? 4. Use profiling to find out what takes the time. Is it CPU-bound? Memory? I/O? can it be fixed in hardware realistically or is it going to be run on a wide variety of systems? 5. Check for better algorithms for the task. quicksort isn't always a good choice. Give lots of examples with good and bad points 6. Check you have a good implementation of your chosen algorithm. they vary wildly. Even when you did it yourself... 7. NOW you can start optimising the code.look at many things, including: temporaries, loops, inlining, lookup tables, caching, ... Remember to go into the things that compilers do and don't do for you, and why you might have problems when you turn on the 'more magic' levels of optimising on bleeding edge compilers...

16 Source Code Browsing

It is a sad but true fact that we programmers have to pick up where others have left off. The main trouble is that the "other guy" did the design by seat-of-the-pants typing and pasting, and he forgot to go back and insert the comments he always intended to do. Now what?

"Read the source, Luke," works, but it is one of the lesser fun things in a programmer's life. To help you through this process are the standard command line tools, '`find`' and '`grep`'. Those two utilities and a little study in the black art of regex will stand you in good stead. For the mortals, there are a couple of powerful UI tools that facilitate this process a great deal. They are:

I mention them both because cscope has been around for over a decade and is simpler to configure and get started with. On the other hand, Source Navigator has a very nice GUI front end to a tool similar to cscope and has numerous additional features. It is a full-fledged development environment that integrates many of the tools we discuss in this book. So, really, the choice is between immediate ('`find`' and '`grep`'), short ('`cscope`') and long term convenience.

This chapter will give you a brief introduction to these tools. Enough of an introduction to make these tools useful to you.

16.1 cscope, a text base navigator

Cscope will help you understand the program by assisting you in finding the usage and definitions of variables and procedures. You supply it with the list of files it needs to search. It then provides search capabilities with a variety of simple methods.

Cscope is curses based (text screen). An information database is generated for faster searches. The fuzzy parser supports C, but is flexible enough to be useful for C++ and Java. It supports a command line mode for inclusion in scripts or as a backend to a GUI or other frontend.

Cscope provides methods of searching for:

- all references to a symbol
- function/procedure invocations
- global symbol definitions
- function/procedure definitions (code)
- a simple text string
- regular expression pattern
- a file
- files that include another file

Once found, you are shown a scrollable menu of places where the item in question was found. You select the the item and your selected editor is started on the line where the selected text was found. When configured to open editor sessions in independent windows, looking through all the places where a variable gets used becomes a humanly manageable task.

16.1.1 special editor features

Several editors have specially written modes for cscope. Among these are: XEmacs, emacs, vim and nvi. Xemacs in particular has some significant features layered on top of the standard cscope features:

- An automatic, hierarchical, search path mechanism exists, for locating cscope index files. If a database isn't found in the current directory, the interface will automatically search parent directories for index files.

- In addition to your basic (normal) cscope setup, the XEmacs interface is also designed to support LARGE projects. Files which are indexed can be spread out over multiple directories, and these directories do NOT have to share a common root directory. Also, cscope index files can be shared amongst users. This is very useful for group software development.
- Multiple cscope index files can be searched. Unlike plain cscope, you're not limited to searching only one database.
- When searching multiple database (index) files, results can be returned from either the first database that contains matches, or all databases that contain matches. This is very useful when you have a local (partial) source tree, yet want to be able to search both your local tree and your project's full source tree.
- Cscope is integrated into the C, C++, and dired modes. Pull-down and pop-up menus exist, as well as normal key bindings.

For those of you who use emacs, there are even special cscope emacs modes that make it easier still.

16.1.2 acquiring and installing

CSCOPE comes with some Linux distributions and it comes with the standard SVr4 development tools. If you do not have it, it can be obtained from its home development site on Source Forge:

```
http://cscope.sourceforge.net
```

It has been fully AUTOCONF-ed, so once you have downloaded and unrolled the tarball, it should take little more than:

```
sh path/to/configure && \
make && \
make check && \
make install && \
```

@xref{wherever}

16.1.3 configuring an editor

CSCOPE displays a menu of search commands. You fill in the search field for one of these commands, and then CSCOPE displays a menu of files and lines that match the criteria. When you pick from this latter menu, CSCOPE tries to fire up your editor or viewer of choice to display the text at the indicated line. It tries to use an environment variable to determine which to use. In order of preference:

CSCOPE_EDITOR	The preferred choice
VIEWER	read-only access implied
EDITOR	common environment variable for showing preference
vi	the viewer used if none are selected as above

A special choice for CSCOPE is actually a good idea. It is very convenient to be able to access the CSCOPE menu while viewing previous results in alternate windows. If you use emacs, this can be accomplished by starting the server process in your main emacs editing session and setting your editor to EMACSCLIENT. You can accomplish a similar functionality with VI by writing a small wrapper script and specifying it as your editor:

```
#! /bin/sh
xterm -e /bin/vi $ &
```

Now, your cscope session will remain active concurrently with your viewer sessions. Be sure to put this shell script in a directory named in the $PATH environment variable and be sure also to make the script executable by typing: '`chmod a+x script-name`'.

16.1.4 simple usage

This will be a very simple example.

- Change directory into your example source directory.
- '`cscope 'find . -name '*.[cChHly]''`'
  This will find all the C, C++, lex and yacc sources in the current directory and below, tell CSCOPE to index them and then present its search menu.

You should now be able to locate and view the definitions and usages of the various symbols in your program(s).

16.2 Source Navigator, a GUI browser

17 State of the World Address

GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and

that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus

accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole

count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.1
> or any later version published by the Free Software Foundation;
> with the Invariant Sections being LIST THEIR TITLES, with the
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

A copy of the license is included in the section entitled "GNU
Free Documentation License".

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being LIST”; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Example Index

This is an alphabetical list of all of the code examples from this book.

A

B

C

D

E

F

G

H

I

L

M

N

P

R

S

T

U

V

Y

Macro Index

This is an alphabetical list of the M4, M4sugar, M4sh, Autoconf and Autotest macros. To make the list easier to use, the macros are listed without their preceding ‘m4_’, ‘AS_’, ‘AC_’ or ‘AT_’.

C

D

I

M

O

P

S

Index

%

A

B

C

D

E

F

G

H

K

L

M

N

O

P

Q

R

S

T

U

Y

Fixme Index

Short Contents