Debugging C and C^{++} code in a Unix environment

J.H.M. Dassen

jdassen@wi.LeidenUniv.nl

I.G. Sprinkhuizen-Kuyper

kuyper@wi.LeidenUniv.nl

Debugging C and C^{++} code in a Unix environment

by J.H.M. Dassen and I.G. Sprinkhuizen-Kuyper

Copyright and Permission Notice

Table of Contents

Abstract

This document describes several techniques and tools for debugging code in C-like languages in a Unix environment.

Chapter 1. Introduction

Debugging is the art of removing bugs from software. The software may be code, documentation, or any other intellectual product. Here, we will look at the debugging of computer programs (or libraries) written in C or C^{++} in a Unix environment. Most of it is also applicable to other compiled procedural and object oriented languages like Pascal, Modula and Objective C.

We will mostly focus on techniques and tools to assist in debugging. Of course, it is better to prevent bugs from slipping into your code in the first place. Sometimes it is difficult to distinguish between good coding practices and good debugging practices, because good debugging practices often involve preparation and prevention. So, we will also discuss some good coding practices that you should consider adopting. These practices will not make your programs bug-free, but they will diminish the occurrence of certain types of bugs, while preparing you better for dealing with the remaining ones.

It is our experience that many people waste large amounts of time on localising bugs that are quite easy to fix once they are found, because they are not aware of, or do not know how to use, the tools, techniques and practices available to them.

Our goal is to help you prevent wasting your time in this fashion. We hope you will invest time to study the material covered here; we are convinced this investment will pay off.

Chapter 2. Conventions

This paper follows some Unix conventions: commands and names of manual pages are written **like this**; for manual pages like this: *ls(1)*, where the section is indicated in parentheses. Also, some of the terminology (‘foo’, ‘bar’, ‘grok’, ‘RTFM’) comes from Unix hackerdom; see [JARGON] if you are interested in it.

Chapter 3. Aspects of debugging C and C^{++} code

Debugging C and C^{++} code entails noticing, localising, understanding and repairing bugs.

Noticing and localising a bug

You might think that noticing a bug is easy: you know what your code should do, and you notice that it does not do that. This easiness is deceptive. Noticing a bug involves testing. Testing is best done in a disciplined fashion, and, wherever possible, in an automated fashion [1] . For certain types of programs (e.g. compilers) it is relatively easy to construct tests (input + expected output/result) and to run these automatically — say, after each build.

You should prepare tests carefully. Make sure that if a test fails, you can see what goes wrong.

In a Unix system, a bug often manifests itself as a program crash, leaving a core dump. In the section called *Core dumps*, we will see what a core dump is, and how it can help you in debugging your code.

Understanding a bug

You should make sure that you understand a bug fully (grok ([JARGON]) it), before you attempt to fix it. Ask yourself the following questions:

- Have I really found the cause of the problem I observed, or is this a mere symptom?
- Have I made similar mistakes (especially wrong assumptions) elsewhere in the code?
- Is this cause just a programming error, or is there a more fundamental problem (e.g. the algorithm is incorrect)?

Repairing a bug

Repairing a bug is more than modifying code. Make sure you *document* your fix properly in the code, and test it properly.

After repairing a bug, ask yourself what you can learn from it:

- How did I notice this bug? This might help you to write a test case to detect it if it slips in again.
- How did I track it down? This will give you better insight in which approach to take in case you encounter similar symptoms again.
- What type of bug was it (see the section called *Types of bugs*)? Do I encounter this type often? If so, what can I do to prevent it from re-occurring?

What you learn is probably valuable not only to you in developing this particular piece of code. Try to communicate what you learned to your colleagues, for instance by writing it down in a pattern-like fashion (e.g. ‘IF you find your program foos bars AND it does not foo bazs THEN try frobbing it’).

Quite often, we find that one of the main reasons why tracking down a bug takes so long, is that we have made unjustified assumptions about parts of our code [2] .

Types of bugs

Experience with bugs shows that they are seldom unique. In deciding how to tackle a particular bug, it is often helpful to attempt classify it.

We will give a very coarse-grained classification; you are encouraged to modify and refine it according to your own insights. We list the categories in order of increasing difficulty (which, luckily, is also in order of diminishing frequency).

- Syntactical errors. These are errors that your compiler should catch. Note the ‘should’: compilers are complex pieces of software, that can be buggy themselves. For example a missing ’;’ or a missing ’}’. Often the place where the compiler remarks the bug is (far) after the place where the bug really is.
- Build errors. Some errors can result from linking object files from before and after a change together. Make sure you use a Makefile, and that it accurately reflects the dependencies involved in building your project. See the section called *An example makefile* in Appendix A for a way to track dependencies automatically.
- Basic semantic bugs, such as using uninitialised variables, dead code [3] and certain type problems. A compiler can often bring these to your attention, but it must be told to do so explicitly (e.g. through warning and optimisation flags [4] ; see the section called *Using the compiler’s features*).
- Semantic bugs, such as using the wrong variable or using ‘&’ ’&&’. No compiler or other tool can find these. You’ll have to do some thinking here. Testing your program step by step using a debugging tool can help you here.

Note that there are many ways of classification, most of which are orthogonal to each other. For example, hackers tend to distinguish between Bohr bugs and Heisenbugs ([JARGON]). Bohr bugs are ‘reliable’ bugs: given a particular input, they will always manifest themselves. Heisenbugs are bugs that are difficult to reproduce reliably; they appear to depend on the phase of the moon (environmental factors like time, particular memory allocation etc.). A Heisenbug is very often the result of errors in pointers: using memory that is not allocated. So use tools (Electric Fence, see the section called *Memory allocation debugging tools*) to check all pointers and array boundaries. (Another cause is the use of uninitialised variables).

C and C^{++} specific problems

There are some features of the C and C^{++} languages and the associated build process that often lead to problems.

Preprocessor

C and C^{++} use a preprocessor to expand macro's, declare dependencies and import declarations and to do conditional compilation. In itself, this is quite reasonable. You should realise however that all of these are done on a textual level. The C/C^{++} preprocessor does *not*

This can make it difficult to track down missing declarations, it can lead to semantic problems because of macro expansion and it can cause subtle problems.

If you suspect a problem due to preprocessing, check out the preprocessor's manual (e.g. [CPP]) and let it expand your file for examination.

Strong systems dependency

C was developed for use as a systems programming language. C and also C^{++} can give you access to a lot of operating system functionality. Unfortunately, there are a lot of small but significant differences among various Unix systems:

- Some system calls are not available on all systems.
- Some system calls and library functions are defined in different header files on different systems.
- There may be differing semantics for particular routines. For example, on Sys V-like systems, a signal handler reinstalled. On BSD-like systems, a signal handler stays in place until explicitly removed.

Also, the size and representation of some of C's and C^{++}'s basic types is dependent on the underlying system. As a C or C^{++} programmer, you should be aware of what things are explicitly undefined in the C or C^{++} standard, and thus are implementation (system or compiler) dependent. There are standard ways to overcome some of these problems, like using `sizeof` instead of the concrete size of the variable on the current system.

Weak type system

C and C++ have a type system, but it is very weak. You can do all kinds of conversions, many of which can be system dependent or meaningless. Also, the compiler can do some implicit conversions that may cause havoc.

Most errors due to the weak type system can be caught in the bud by doing static analysis early; see the section called *Using the compiler's features*.

Explicit storage allocation and deallocation

In C and C++, you have to explicitly allocate and deallocate dynamic storage through `malloc` and `free` (for C) and through `new` and `delete` (for C++). If memory (de)allocation is done incorrectly, it can cause problems at run time such as memory corruption and memory leaks (the memory use of a program keeps on increasing during execution).

Common errors are:

- Trying to use memory that has not been allocated yet.
- Trying to access memory that has been deallocated already.
- Deallocating memory twice.

These errors are difficult to correct without using proper tools; see the section called *Memory allocation debugging tools*.

Name space pollution

In C and C++ programmers commonly do not to try to prevent name space pollution (name conflicts).

- Use the `static` keyword to indicate functions and variables whose scope is restricted to the current file.
- Use as few global variables and functions as necessary. If you have to use a large number of them, prefix their names consistently (e.g. `MYPROJECT_someglobal`).

Incremental building/linking

C and C^{++} code can be built incrementally; usually **make** is used to specify dependencies among files for a build. If a Makefile does not specify dependencies properly, you can end up with executables linked to old versions of modules which can be buggy or incompatible with recently introduced changes in other modules.

The build process

Bugs you encounter may not be due to your C or C^{++} code; they might be the result of how your executable/library was built. Make sure that you understand how the build process is organised.

You should use a Makefile. A Makefile describes how to build your project: it lists the files involved in your project, their interdependencies and how a tool should build intermediary files and the end product. Make sure you have listed all dependencies; missing even a single dependency can lead to subtle problems.

make is a powerful tool, and it pays off to acquaint yourself with it well. For instance, in general you should not list compilation lines directly. GNU **make** has some builtin rules (so called *implicit rules*) on how, say, `.o` files are built from `.c` files. To use those rules, you only specify the dependencies (e.g. `foo.o: foo.c foo.h bar.h` (for C or `foo.cc` for C^{++} programs)), and no build rule. The implicit rules have a number of variables that you can set (e.g. `CC` for the C or C^{++} compiler, `CFLAGS` for the compilation flags, `LOADLIBES` for the libraries). Using the implicit rules makes your makefiles shorter, easier to read and easier to modify. See [MAKETUT], [MAKETUT2] and [MAKE] for details on **make**.

For C and C^{++}, the programs involved in building and running programs are:

preprocessor

The preprocessor's main task is to process (header (`.h`)) file inclusions and macros; it outputs pure C or C^{++} code.

compiler

The compiler translates pure C or C^{++} code to assembly language.

assembler

The assembler translates assembly code to binary object code (`.o`).

linker

> The linker combines a number of object files and libraries to produce an executable or library. If this executable or library needs no external libraries, it is called *statically linked*; otherwise it is called *dynamically linked*.

dynamic loader

> The dynamic loader’s task is to load the libraries (or library parts) required by a dynamically linked executable prior to actually running that executable.

Core dumps

A *core dump* is a snapshot of the execution of a program at the moment it is aborted by the operating system (e.g. for attempting to violate the memory protection). A normal core dump is not very helpful unless you are an expert. In the section called *The debugger*, we will see how to make core dumps more helpful for debugging.

By default, core dumps do not contain all the information you’d like them to. For example, a core dump can tell you that you where dereferencing a pointer at memory location 0x12345 while executing the instruction at 0x45678. You’d probably like to see a message that means more to you (‘The program was aborted while attempting to dereference foo, which was NULL, at bar.c line 23’). This is possible, but it requires you to include such information in advance.

Also, note that a core dump is a snapshot; it does not include the history of how your program came to the problematic state. What a core dump shows you is a manifestation of a bug; the point where a program dumps core is not always the location of the bug itself, which may be located 100000 instructions back in time. Often, you can reconstruct the history of a run from a core dump, but this is difficult. printf debugging (see the section called *printf() debugging*) and possibly system call tracing (see the section called *System call tracers*) are useful techniques to do this. Using a debugger (see the section called *The debugger*) is advised.

Debugging techniques

In this section a number of debugging techniques from reading manuals to using tools are described.

Using the compiler’s features

A good compiler can do a good deal of *static analysis* of your code: the analysis of those aspects of a piece of code that can be studied without executing that code.

Static analysis can help in detecting a number of basic semantic problems such as type mismatches and dead code.

For gcc (the GNU C compiler) there are a number of options that affect what static analysis gcc does and what results will be shown. There are two types of options:

Warning options

gcc has a great number of warning flags. Most have the form `-W`*phrase*. You should pick ones relevant to you at the start of coding and put them into your Makefile (use the implicit rules, and put them in the `CFLAGS` variable). Note that `-Wall` does *not* switch on all warnings. It enables a set of warnings that gcc’s developers consider useful under nearly all circumstances. In addition to `-Wall` we recommend at least the following warnings when writing new code: `-Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wstrict-prototype` [5] As an example: The following code will result in a warning because the possibility exists that the function returns without returning a value: `foo(int a) { if (a > 0) return a; }`

Optimisation flags

gcc also supports a number of optimisations. Some of these trigger gcc to do extensive flow analysis of your code, resulting in for example dead code removal. For normal use, we recommend `-O2`. Do not use higher optimisation levels unless you know what you are doing; the higher levels can contain experimental optimisations which could generate bad code. Also note that on some systems, enabling optimisation makes debugging using a debugger virtually impossible.

For full documentation of these options, see the chapter ‘GNU CC Command Options’ in [GCC].

The RTFM technique

RTFM stands for *Read The Fine Manual*. Make sure you take the time to find relevant documentation for the task at hand, i.e. the documentation of the tools (not only the compiler, but also **make**, the preprocessor and the linker), libraries and algorithms you are expected to use, such as [CPP][GCC][MAKE]. Often you do not need to know everything in the documentation, but you do need to be aware what documentation is relevant and what its purpose is. You should at the very least browse through it; hopefully this will give you a feeling of *deja-vu* where needed, so you know where to look.

In examining documentation, you should distinguish between tutorials and reference documentation.

A tutorial is a document intended to teach, mostly by example. In a tutorial, conveying ideas is often more important than literal truth. For example, this document is a tutorial.

Reference documentation assumes that you are familiar with its topic, and that you are looking for a definite answer to a specific question. Good reference documentation is exhaustive and enables you to find your answer quickly, through meta-information, such as a table of contents, an index (often several ones) and cross-references. Online hypertext is a convenient format for reference documentation.

Make sure that your reference documentation is up to date and accurate. Never mistake a tutorial document for a reference document; tutorials can never (and thus, should never) be used as authoritative documentation.

In the section called *Documentation formats* in Appendix A, we discuss a number of different documentation formats and how to handle them. Especially the Info documentation and the man-pages are very important.

printf() debugging

printf debugging is our term for a debugging technique we encounter all too often. It consists of ad hoc addition of lots of `printf` (C) or `cout` (C^{++}) statements to track the control flow and data values in the execution of a piece of code.

This technique has strong disadvantages:

- It is very ad hoc. Code is temporarily added, to be removed as soon as the bug at hand is solved; for the next bug, similar code is added etc. There are better ways of adding debugging information, as you shall see shortly.
- It clobbers the normal output of your program, and slows it down considerably.
- Often, it does not help. Output to `stdout` is buffered, and the contents of the buffer are usually lost in case of a crash. Thus, you'll most likely miss the most important information, causing you to start looking in all the wrong places [6] .

If you consider using printf debugging, please check out the use of assertions (see the section called *Assertions: defensive programming*) and of a debugger (see the section called *The debugger*); these are often much more effective and time-saving.

There are some circumstances where printf debugging is appropriate. If you want to use it, here are some tips:

- Produce your output on `stderr`. Unlike `stdout`, `stderr` is unbuffered. Thus using `stderr`, you're much less likely to miss the last information before a crash.

- Do not use `printf` directly, but define a macro around it, so that you can switch your debugging code on and off easily.
- Use a debugging *level*, to manage the amount of debugging information.

Here is a nice way to do it. File `debug.h`:

```
#ifndef DEBUG_H
#define DEBUG_H
#include <stdarg.h>

#if defined(NDEBUG) && defined(__GNUC__)
/* gcc's cpp has extensions; it allows for macros with a variable number of
   arguments. We use this extension here to preprocess pmesg away. */
#define pmesg(level, format, args...) ((void)0)
#else
void pmesg(int level, char *format, ...);
/* print a message, if it is considered significant enough.
      Adapted from [K&R2], p. 174 */
#endif

#endif /* DEBUG_H */
```

File `debug.c`:

```
#include "debug.h"
#include <stdio.h>

extern int msglevel; /* the higher, the more messages... */

#if defined(NDEBUG) && defined(__GNUC__)
/* Nothing. pmesg has been "defined away" in debug.h already. */
#else
void pmesg(int level, char* format, ...) {
#ifdef NDEBUG
/* Empty body, so a good compiler will optimise calls
   to pmesg away */
#else
        va_list args;

        if (level>msglevel)
                return;

        va_start(args, format);
        vfprintf(stderr, format, args);
        va_end(args);
#endif /* NDEBUG */
#endif /* NDEBUG && __GNUC__ */
}
```

Here, `msglevel` is a global variable which you have to define, that controls how much debugging output is done. You can then use `pmesg(100, "Foo is %l\n", foo)` to print the value of foo in case `msglevel` is set to 100 or more.

Note that you can remove all this debugging code from your executable by adding `-DNDEBUG` to the preprocessor flags: for GCC, the preprocessor will remove it, and for other compilers `pmesg` will have an empty body, so that calls to it can be optimised away by the compiler. This trick was taken from `assert.h`; see the next section.

Assertions: defensive programming

If you take a careful look at your code, you'll notice that in every part (say, function or loop) you make a lot of assumptions about the other parts.

Say you write your own power function [7] that takes an integer argument `e`, but implicitly assumes this argument is positive [8] .

assertions are expressions that should evaluate to *true* at a specific point in your code; well-known examples are pre- and post-conditions for functions. If an assertion fails to evaluate to *true*, you have found a problem (possibly in the assertion, but more likely in your code). It makes no sense to execute after an assertion failure.

Writing down assertions means making your assumptions explicit. In C and C++, you can `#include<assert.h>`, and write the expression you want to assert as the argument to `assert`, e.g. `assert(e > 0)`. See *assert(3)*.

With the `assert` macro, your program will be aborted as soon as an assertion fails, and you will get a message stating that the assertion *expression* failed at line *l* of file *f*.

`assert` is a macro; you can remove all assertion checking from your executable by compiling it `-DNDEBUG`. You should of course do this only when you release your program to users (and even then only when you are convinced there are no more bugs in it, or when execution speed is of primary concern).

ANWB debugging

'ANWB debugging' is based on a simple principle: the best way to learn things is to teach them.

In 'ANWB debugging' you find a, preferably innocent and willing, bystander and explain to her how your code works [9] . This forces you to rethink your assumptions, and explain what is really happening; often you find the cause of your problems this way.

Code grinding (code walk through)

A similar technique to ANWB debugging is to print your code, leave your terminal, and go to the cafetaria and do some serious caffeine and sugar intake while reading (and annotating) your code carefully.

Tools

In this section a number of tools relating to debugging and analysing your programs are described.

The editor

Using an editor suitable for coding can make life easier for you. A good programmer's editor should offer features like

- Syntax highlighting
- Show matching braces
- Automatic indentation
- Easy navigation (like vi's and emacs' tags)
- Easy compiling (e.g. vim's `:make` or emacs' compile)

A version management system

Even for small programming jobs, it is useful to archive your source code (including associated makefiles, scripts, documentation etc.) using a version management system like [RCS].

For large programming projects (e.g. EGCS (http://egcs.cygnus.com), the Experimental GCC Compiler Suite, GNOME (http://www.gnome.org), the GNU Object Model Environment, and Mozilla (http://www.mozilla.org)) version management is essential. Such projects often use **CVS** ([CVS]), a version management system with parallel development and internetworking support.

The debugger

A *debugger* is a program that allows you to peep into a program's execution or to do a post-mortem from a core dump. It can answer questions like

- At what point did the program crash? Through which function calls did it get there?
- What was the value of `foo` at the time of the crash?

and it can be used to

- step through your code
- interrupt execution at specific points in your code
- break execution on a specific condition

In order to make effective use of a debugger, you should compile the code you want to debug with debugging information enabled. This will allow the debugger to translate machine addresses and values to a human-readable form.

For the GNU tools, this means you should give the `-g` or `-ggdb` options to **gcc**.

In some environments, you cannot debug optimised code; in others, you can, but it can be confusing. If you cannot use a debugger on optimised code in your environment, do not use optimisation.

gdb is the GNU debugger. It is quite powerful, but unfriendly because of it's terminal interface. If it is available to you, we recommend using a front end for **gdb**, such as **ddd** ([DDD]) or **xxgdb**. Such a front end does not add to the power of the debugger, but makes it much easier to use. Especially, **ddd** is very useful if you need to visualise a data structure.

Consider the program `buggy.cc`:

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

typedef struct elt {
int data;
struct elt* next;
} elt;
```

```
int main(int argc, char* argv[]) {
int i;
elt* list = NULL;
elt* p = NULL;
for (i = 1; i < argc ; i++) {
p = (elt *) malloc(sizeof(elt));
p->data = (int) argv[i][1]; /* Fails with empty argument */
p->next = list;
list = p;
}
while(list != NULL) {
p = list;
list = list->next;
free(p->next); /* Oops... should be p */
}
exit(0);
}
```

Using Electric Fence (see next subsection) it is possible to force a segmentation fault on this code. For plain **gdb**, here's a short example session:

```
ultra5 jdassen 16:30 /home/kuyper/debug > g++ -g buggy.cc -lefence
ultra5 jdassen 16:30 /home/kuyper/debug > gdb ./a.out
GNU gdb 4.17
Copyright 1998 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 "sparc-sun-solaris2.5"...
(gdb) run
Starting program: /home/kuyper/debug/./a.out
Program exited normally.
(gdb) set args foo bar baz
(gdb) run
Starting program: /home/kuyper/debug/./a.out foo bar baz
  Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens.
Program received signal SIGSEGV, Segmentation fault.
0x10d30 in main (argc=4, argv=0xeffff3e4) at buggy.cc:21
list = list->next;
(gdb) where
#0  0x10d30 in main (argc=4, argv=0xeffff3e4) at buggy.cc:21
(gdb) p list
```

```
$1 = (elt *) 0xef507ff8
(gdb) p *list
$2 = {data = 97, next = 0xef503ff8}
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

Memory allocation debugging tools

As discussed earlier in the section called *C and C^{++} specific problems*, one of the causes of problems with C and C^{++} code, is the policy of requiring explicit allocation and deallocation of dynamic storage (*malloc(2)* and *free(2)* (C) or `new` and `delete` (C^{++})). As C and C^{++} rely heavily on pointers, this is a very common cause of problems.

Pointer problems can be quite difficult to detect, as they often show up far away from the cause of a problem. Fortunately, there are a number of tools to help you deal with them; see the section called *Memory allocation debugging tools* for an extensive list.

We will focus on memory corruption and incorrect pointer use (i.e. those problems that cause core dumps and strange behaviour) rather than on memory leaks. Memory leaks are not a problem in most cases. They waste resources, but often this is only a problem with software that should run continuously (such as X servers and Unix daemons) [10] . The discussion also applies to related problems, such as dealing with out of bounds addressing of arrays.

One memory allocation debugging tool we recommend is Electric Fence by Bruce Perens. It is a library that overrides the default `malloc` and `free` and `new` and `delete`. By default, it allocates memory at the end of a memory page (which makes it memory hungry). This strongly increases the chances of an OS trap as soon as an attempt is made to address memory not allocated through `malloc` (e.g. attempting to access memory that has been `free`d already, or following a pointer into hyper-space). You can use it by simply adding `-lefence` to your link line. Running the program with **ddd** (or **gdb**) and investigating the calls where the segmentation faults occur, helps you to find the memory errors causing code (pointers or array bounds). See *efence(3)* for documentation. See also the example in the previous section.

Another tool is Checker. The Checker tool uses it's own version of **gcc**, **checkergcc** to include boundary checks in your code. It is probably better than Electric Fence, but less easy to use as it requires all libraries you use to be compiled with **checkergcc** themselves.

System call tracers

A *system call tracer* is a program that allows you to see what system calls (including parameters and return values) a process makes. It allows you to examine problems at the boundary between your code and the operating system. In a Unix system, programs cannot access the hardware or the network directly; all such access takes place through library calls. Also, most resource allocation (e.g. memory, semaphores etc.) is done through library calls.

You can run a system call tracer on a regular binary. Thus, it can be a useful tool if some proprietary binary-only program, say Netscape, gives an uninformative error message, like ‘file not found’. You can use a system call tracer to see what files Netscape tries to access, and whether or not it succeeds in accessing them.

Because a system call tracer has no knowledge about the origin of binaries, it cannot tell you where in your code a certain system call is made; you will have to reconstruct this from the trace, or use printf debugging (see the section called *printf() debugging*).

Under Linux, you can use *strace(1)*. Most system call tracers can be attached to a running process, so you do not have to trace the whole execution of a program.

Profilers

A profiler is a tool that administrates the times of the running program needed for each procedure call. Such a tool is very important when the efficiency of a program has to be improved. In a unix environment the profiler **gprof** can be used when compiling with **gcc** or **g++** using the option `-gp`. Running the program will result in a file `gmon.out`, which will be used by **gprof** to give information about the time spent in different routines. See the manpages of **gprof**.

Conclusions

Try to prevent bugs by analysing the problem before and by designing good program structures. However, it is always possible that a program contains a number of bugs. To find these bugs the following methods exist:

- always be aware of the RTFM technique
- use the possibilities of warnings the compiler can give you
- using a debugging tool (**ddd** or **gdb**) is advisable above ‘`printf`() debugging’
- add assertions at critical places of your program
- use Electric Fence (or an other memory allocation debugging tool) to find problems with pointers and array bounds

Bibliography

The GNU C preprocessor This comes with the gcc source in TeXinfo format, and is usually installed in 'info' format.

Brian Berliner *CVS II: Parallelizing Software Development* Included in the CVS source distribution.

Using and Porting GNU CC The gcc and g++ manual; it comes with the gcc source in TeXinfo format, and is usually installed in 'info' format.

The GNU make manual This comes with the GNU make source in TeXinfo format, and is usually installed in 'info' format.

Paul Haldane *Make (ftp://ftp.ed.ac.uk/pub/courses/make.tex)*

How to write a Makefile (http://vertigo.hsrl.rutgers.edu/ug/make_help.html)

The Jargon file (http://www.tuxedo.org/~esr/jargon/)

Dorothea Lütkehaus and Andreas Zeller *DDD — the Data Display Debugger (http://www.cs.tu-bs.de/softech/ddd/)*

Brian W. Kernighan and Dennis M. Ritchie *The C programming language, second edition* Prentice-Hall, 1988.

Ben Zorn *Debugging Tools for Dynamic Storage Allocation and Memory Management (http://www.cs.colorado.edu/homes/zorn/public_html/MallocDebug.html)*

Walter F. Tichy *RCS — A System for Version Control* included with the RCS source distribution.

Notes

1. Unix's powerful scripting abilities come in very handy in this case.
2. One of Ray's experiences: Once, a memory allocation tool told me that there was a problem with my own string routines. I did not believe it. Some months later, I spent hours in tracking down a nasty bug to find it was an off-by-one error in my string copy routine, precisely where the tool complained.
3. Dead code is code that will never be used. Often it is a bug, sometimes it can be an example of defensive programming: 'if this part of the code is reached then something is terribly wrong, thus give a warning on the screen and stop this program'. Also automatically generated code often contains dead code. The GNU CC does not give warnings for unreachable code. However, it will be removed during optimisation.
4. To optimise, a compiler has to do some quite sophisticated analysis, including data-flow analysis, which can detect dead code.
5. See the gcc man-pages or info about the meaning of these warnings.
6. It is possible to force the buffers to be emptied by using the function `flush`, or C++'s << `endl`.

7. Bad example. You should use *pow(3)*, which is probably better tested and faster.
8. Even if this is documented in comments, we consider it here as implicit, because no tool can determine or check this for you automatically.
9. The name derives from the ANWB, the Dutch organisation that helps with car trouble. They maintain a communication system along the highways. In the extreme case you can’t find anyone willing, you could consider going to a highway and use them to explain your problems to.
10. Localising a memory leak is quite difficult. We recommend using Purify if you have access to it; we are not aware of a freeware tool that comes close to it for tracking memory leaks. In C^{++} memory leaks can be prevented by carefully writing and using class destructors: they should free all memory allocated by the class in destruction.

Appendix A.

An example makefile

Here is an example Makefile that illustrates advanced features of make (e.g. automatic dependency tracking; implicit rules):

```
# An example Makefile

# The C++ compiler - this variable is used in the implicit rule for producing
# N.o from N.cc (or N.C)
CXX = g++

# The C compiler - this is used in two implicit rules:
# - the rule for producing N.o from N.c
# - the rule to link a single object file.
CC = gcc

# Preprocessor flags
CPPFLAGS =  # -DNDEBUG when the debugging code as given in section 8.3
            # has to be turned of

# We split up the compiler flags for convenience.

# Warning flags for C programs
WARNCFLAGS = -Wall -W -Wshadow -Wpointer-arith -Wbad-function-cast \
-Wcast-qual -Wcast-align -Wstrict-prototypes -Wmissing-prototypes \
-Wmissing-declarations

# Warning flags for C++ programs
WARNCXXFLAGS = $(WARNCFLAGS) -Wold-style-cast -Woverloaded-virtual
# Debugging flags
DBGCXXFLAGS = -g
# Optimisation flags. Usually you should not optimise until you have finished
# debugging, except when you want to detect dead code.
OPTCXXFLAGS = # -O2

# CXXFLAGS is used in the implicit rule for producing N.o from N.cc (or N.C)
CXXFLAGS = $(WARNCXXFLAGS) $(DBGCXXFLAGS) $(OPTCXXFLAGS)

# The linker flags and the libraries to link against.
# These are used in the implicit rule for linking using a single object file;
# we’ll use them in our link rule too.
LDFLAGS = -g
# Use Electric Fence to track down memory allocation problems.
LOADLIBES = -lefence

# The program to build
PROGRAMS = ourprogram

# All the .cc files
SOURCES = $(wildcard *.cc)
# And the corresponding .o files
OBJECTS = $(SOURCES:.cc=.o)


# The first target in the makefile is the default target. It’s usually called
# "all".
all: depend TAGS $(PROGRAMS)

# We assume we have one program (‘ourprogram’) to build, from all the object
# files derived from .cc files in the current directory.
ourprogram: $(OBJECTS)
$(CC) $(LDFLAGS) -o ourprogram $(OBJECTS) $(LOADLIBES)

# "clean" removes files not needed after a build.
clean:
rm -f $(OBJECTS)

# "realclean" removes everything that’s been built.
realclean: clean
rm -f $(PROGRAMS) .depend TAGS

# "force" forces a full rebuild.
force:
$(MAKE) realclean
$(MAKE)

# "depend" calculates the dependencies.
depend:
rm -f .depend
$(MAKE) .depend

# This is the actual dependency calculation.
.depend: $(SOURCES)
rm -f .depend
# For each source, let the compiler run the preprocessor with the -M and -MM
# options, which causes it to output a dependency suitable for make.
for source in $(SOURCES) ; do \
  $(CC) $(CPPFLAGS) -M -MM $$source | tee -a .depend ; \
done

# Include the generated dependencies.
ifneq ($(wildcard .depend),”)
include .depend
endif

# Produce a ‘TAGS’ file for (x)emacs to use for browsing the source code.
TAGS: $(SOURCES)
etags -o TAGS -typedefs-and-c++ $(SOURCES)


# Tell make that "all" etc. are phony tar-
gets, i.e. they should not be confused
# with files of the same names.
.PHONY: all clean realclean force depend
```

Documentation formats

Under Unix, there are several formats for documentation in common use.

Manual pages

Online manual pages (‘man-pages’) are the traditional format for documentation under Unix. Man-pages are almost always reference documentation, documenting exhaustively switches and parameters. Man-pages are written in nroff format (a typesetting language), and mostly viewed as ASCII text with simple markup (bold, italic).

You can search through a man-page using the search facility of your pager. We recommend **less** rather than the default **more**.

If you have the source version of a manual page, you can make a nice printed version with a command sequence like `groff -Tps -mandoc /usr/man/man1/ls.1 | lpr`

If you have only the formatted version (‘catman’) of a manual page, you can try printing that; if that does not work, you can filter it to get plain ASCII with **col**: `col -b < /var/catman/cat5/shells.5 | lpr`

Info documentation

The ‘info’ format is the preferred format of the Free Software Foundation. Info is an old hypertext format which you can read through **info**. The info reader is not very comfortable, but has a good search facility, and it does not require X. However, using xemacs for reading info makes life easier.

‘info’ format is not written directly; it is produced from files in the TeXinfo [1] format (these files are not installed on most systems; you’ll probably have to unpack source archives to find them). You can produce nice printable versions through something like `texi2dvi make.texi | dvips -f | lpr`

HTML and PDF

HTML (HyperText MarkupLanguage) is the default format of information on the World Wide Web; its hypertext facilities and the nice browsers for it make it increasingly popular as a documentation format. It is however difficult to search through a number of HTML files with most browsers. X is not required for reading HTML; there are text-mode browsers available for it (e.g. **lynx**), and it can even be read without a browser.

PDF (Portable Document Format) is a successor to PostScript with hypertext facilities. Unlike HTML, it provides the author full control of the layout of a document. You need a windowing system to read PDF.

Flat ASCII, DVI, PostScript etc.

Many programs come with documentation in ‘unstructured’ formats, such as plain ASCII files (e.g. READMEs), (La)TeX files and PostScript files. Because of their unstructuredness, they are difficult to search through. Also, documentation in these formats is often not installed; you’ll have to get it from the source.

You can view documentation in DVI format (TeX’s output format) through **xdvi**, and you can view PostScript through **gv**, **ghostview**, **pageview** or **gs**.

Notes

1. TeXinfo is a set of TeX macros specifically aimed at producing high-quality printouts, while maintaining readability on simple text terminals.