Software debugging techniques

P. Adragna
Queen Mary, University of London

Abstract
This lecture provides an introduction to debugging, a crucial activity in every developer's life. After an elementary discussion of some useful debugging concepts, the lecture goes on with a detailed review of general debugging techniques, independent of any specific software. The final part of the lecture is dedicated to analysing problems related to the use of C++ , the main programming language commonly employed in particle physics nowadays.

Introduction

According to a popular definition [1], debugging is *a methodical process of finding and reducing the number of bugs, or defects, in a computer program*. However, most people involved in spotting and removing those defects would define it as an art rather then a method.

All bugs stem from a one basic premise: something thought to be right, was in fact wrong. Due to this simple principle, truly bizarre bugs can defy logic, making debugging software challenging. The typical behaviour of many inexperienced programmers is to freeze when unexpected problems arise. Without a definite process to follow, solving problems seems impossible to them. The most obvious reaction to such a situation is to make some random changes to the code, hoping that it will start working again. The issue is simple: the programmers have no idea of how to approach debugging [2].

This lecture is an attempt to review some techniques and tools to assist non-experienced programmers in debugging. It contains both tips to solve problems and suggestions to prevent bugs from manifesting themselves. Finding a bug is a process of confirming what is working until something wrong is found. Therefore, an algorithm good in every situation should not be expected: there is no silver bullet for debugging. Experience and ingenuity are part of the quest for bugs, but also disciplined usage of tools.

The importance of a method of finding errors and fixing them during the life-cycle of a software product cannot be stressed enough. Testing and debugging are fundamental parts of programmer's everyday activity but some people still consider it an annoying option. When not carried out properly, consequences can be dreadful. In 1998, for example, a crew member of the guided-missile cruiser USS Yorktown mistakenly entered a zero as data value, which resulted in a division by zero. The error cascaded and eventually shut down the ship's propulsion system. The ship was dead in water for several hours because a program didn't check for valid input [3].

Bugs can also be very expensive. In 1999, the 125 million dollars Mars Climate Orbiter was assumed lost by officials at NASA. The failure responsible for loss of the orbiter was attributed to a failure of NASA's system engineering process. The process did not specify the system of measurement to be used on the project. As a result, one of the development teams used Imperial measurement while the other used the metric system. When parameters from one module were passed to another, during orbit navigation correction, no conversion was performed, resulting in the loss of the craft [4].

These two famous bugs, as others in history of software [5], should make the reader understand the importance of finding errors in software: it is not just an unavoidable part in the development cycle but vital part of every software system's lifespan.

The lecture starts with a general introduction to debugging containing some useful concepts for programmers approaching this subject for the first time. It goes on with a detailed review of general debugging techniques, not bound to any specific kind of software. Since C++ is the main language commonly employed in particle physics nowadays, the final part is dedicated to analysing problems related to the use of this programming language.

The lecture contains several examples and source files, written in C++. They can be compiled and run and provide a starting point for personal experimentation. Although the examples refer very often to a Unix-like operating system, the underlying concepts and techniques are platform independent.

1 General concepts about debugging

After many days of brainstorming, designing and coding, the programmer finally have a wonderful piece of code. He compiles it and runs it. Everything seems pretty straightforward but unfortunately it doesn't work! And now? Now the great fun starts! Time to dig into the wonderful world of debugging.

Despite being the realm of ingenuity and uncertainty, a debugging process can be divided into four main steps:

1. localising a bug,
2. classifying a bug,
3. understanding a bug,
4. repairing a bug.

1.1 Localising a bug

A typical attitude of inexperienced programmers towards bugs is to consider their localisation an easy task: they notice their code does not do what they expected, and they are led astray by their confidence in knowing what their code should do. This confidence is completely deceptive because spotting a bug can be very difficult. As it was explained in the introduction, all bugs stem from the premise that something thought to be right, was in fact wrong. Here is a very simple example of a possible problem.

Listing 1: Bad naming convention causing an endless loop

```
// An example of a problem of scope
void c (void); // function prototype
int x = 1; // global variable
int main()
{
  int x = 5; // local to main
  // Some other code
  while (x < 100)
    c(); // c() uses global
  // Some other code
  return 0;
}
void c(void)
{
  //Some other code
  x *= 10;
  //Some other code
}
```

This program contains a typical endless loop. The main function consists of a loop calling the function **c()** as long as the variable x is lower than 100. **c()** is supposed to increment the variable x, defined globally. Unfortunately, due to a poor naming convention, a new declaration in the main scope cause x not to be incremented as expected, producing an endless loop. This example also illustrates the danger of giving to variables in different scopes the same names: despite language standards, this can be a big source of troubles.

Noticing a bug implies testing. Testing should be performed with discipline and, when possible, automatically, for example after each build of the code. In case of a test failure, the programmer must be able to see what went wrong easily, so tests must be prepared carefully. This lecture will not cover the basic of testing. Information on automatic software testing can be found in [6], [7]. Learning the basics of automatic software testing is highly recommended.

1.2 Classifying a bug

Despite the appearance, bugs have often a common background. This allows to attempt a quite coarse, but sometimes useful, classification. The list is arranged in order of increasing difficulty (which fortunately means in order of decreasing frequency).

Syntactical Errors should be easily caught by your compiler. I say "should" because compilers, beside being very complicated, can be buggy themselves. In any case, it is vital to remember that quite often the problem might not be at the exact position indicated by the compiler error message.

Build Errors derive from linking object files which were not rebuilt after a change in some source files. These problems can easily be avoided by using tools to drive software building, like GNU Make.

Basic Semantic Errors comprise using uninitialised variables, *dead code* (code that will never be executed) and problems with variable types. A compiler can highlight them to your attention, although it usually has to be explicitly asked through flags (cp. 2.1).

Semantic Errors include using wrong variables or operators (e.g., `&` instead of `&&` in C++). No tool can catch these problems, because they are syntactically correct statements, although logically wrong. A test case or a debugger (see par. 2.8) is necessary to spot them.

A funny "physical" classification distinguishes between *Bohrbugs* and *Heisenbugs*. Bohrbugs are deterministic: a particular input will always manifest them with the same result. Heisenbugs are random: difficult to reproduce reliably, since they seem to depend on environmental factors (e.g. a particular memory allocation, the way the operating system schedules processes, the phase of the moon and so on). In C++ a Heisenbug is very often the result of an error with pointers (cp. 3.3).

1.3 Understanding a bug

A bug should be fully understood before attempting to fix it. Trying to fix a bug before understanding it completely could end in provoking even more damage to the code, since the problem could change form and manifest itself somewhere else, maybe randomly. Again, a typical example is memory corruption: if there is any suspect memory was corrupted during the execution of some algorithm, all the data involved in the algorithm must be checked before trying to change them. More about memory corruption is presented in 3.3.

The following check-list is useful to assure a correct approach to the investigation:

- do not confuse observing symptoms with finding the real source of the problem;
- check if similar mistakes (especially wrong assumptions) were made elsewhere in the code;
- verify that just a programming error, and not a more fundamental problem (e.g. an incorrect algorithm), was found.

1.4 Repairing a bug

The final step in the debugging process is bug fixing. Repairing a bug is more than modifying code. Any fixes must be documented in the code and tested properly. More important, learning from mistakes is an effective attitude: it is good practice filling a small file with detailed explanations about the way the bug was discovered and corrected. A check-list can be a useful aid.

Several points are worth recording:

- how the bug was noticed, to help in writing a test case;
- how it was tracked down, to give you a better insight on the approach to choose in similar circumstances;
- what type of bug was encountered;
- if this bug was encountered often, in order to set up a strategy to prevent it from recurring;
- if the initial assumptions were unjustified; this is often the main reason why tracking a bug is so time consuming.

2 General debugging techniques

As said before, debugging is often the realm of ingenuity and uncertainty. Yet a number of tricks can be adopted in the daily programming activity to ease the hunt for problems.

2.1 Exploiting compiler features

A good compiler can do some *static analysis* on the code. Static code analysis is the analysis of software that is performed without actually executing programs built from that software. Static analysis can help in detecting a number of basic semantic problems, e.g. type mismatch or dead code.

Having a look at the user manual of the compiler employed, where all the features should be documented, is highly recommended. For *gcc*, the standard compiler on GNU/Linux systems, there are a number of options that affect what static analysis can be performed. They are usually divided into two classes: warning options and optimisation flags. As far as warning options are concerned, here is a list of useful ones:

Wall enables all the warnings about constructions that some users consider questionable and easy to avoid. This also enables some language-specific warnings;

Wshadow warns whenever a local variable shadows another local variable, parameter or global variable or whenever a built-in function is shadowed. With this flag, the bug shown in listing 1 could have been easily avoided;

Wpointer-arith warns about anything that depends on the "size of" a function type or of `void`;

Wcast-qual warns whenever a pointer is cast so as to remove a type qualifier from the target type, for example if a `const char*` is cast to an ordinary `char*`. Removing qualifier can be a painful source of troubles;

Wcast-align warns whenever a pointer is cast such that the required alignment of the target is increased, for example if a `char*` is cast to an `int*` on machines where integers can only be accessed at two- or four-byte boundaries;

Wstrictprototype this option is valid only for programs written in C and warns if a function is declared or defined without specifying the argument types.

Maintaining the flag **Wall** active at all times is recommended, while the others are especially suitable when compiling new code.

Compilers support also a number of optimisations. Some of these trigger the compiler to do extensive code flow analysis, removing dead code. Nevertheless programmers must understand that optimisation works, to some extent, against debugging. Optimisation implies a lot of code flow analysis, which ends up in rearranging code statements. It means that once optimised, the code could be different from what was originally written, making debugging virtually impossible. Therefore, optimisation flags should be turned on only when the code appears to be reasonably bug free.

As far as gcc is concerned, optimisation level is identified by a number. For standard use, an optimisation level not higher then 2 is enough, since higher levels could contain experimental optimisation which could generate bad code.

For more information about the various options supported by gcc, consult gcc manual [8].

2.2 Reading the right documentation

This seems quite an obvious tip, but too often unexperienced programmers read the wrong papers looking for hints about the task they have to accomplish. The relevant documentation for the task, the tools, the libraries and the algorithms employed must be at fingertips to find the relevant information easily.

As far as documentation is concerned, the most important distinction is between tutorials and references. A tutorial is a pedagogical paper, usually with plenty of examples. It doesn't assume any previous knowledge of the topic and its first aim is to convey ideas about the subject. Reference manuals, on the contrary, are comprehensive and exhaustive descriptions, which allow to find the answers to questions through indexes and cross-references.

In the world of programming, all these types of documents are usually in electronic format. The reference documentation must be up to date, accurate and corresponding to the problems and tools used: looking up in a wrong reference manual could end up in trying to use a feature that is not supported by the current version of the tool, for example.

2.3 The abused cout debugging technique

The cout technique takes its names from the C++ statement for printing on the standard output stream (usually the terminal screen). It consists of adding print statements in the code to track the control flow and data values during code execution. Although it is the favourite technique of all the novices, it is unbelievable how many experienced programmers still refuse to evolve and abandon this absolutely time-wasting and very ad-hoc method.

Despite its popularity, this technique has strong disadvantages. First of all, it is very ad-hoc, because code insertion is temporary, to be removed as soon as the bug is fixed. A new bug means a new insertion, making it a waste of time. In debugging as well as in coding, the professional should aim to find reusable solutions whenever possible. Printing statements are not reusable, and so are deprecated. As we will see shortly, there are more effective ways to track the control flow through messages. In addition, printing statements clobber the normal output of the program, making it extremely confused. They also slow the program down considerably: accessing to the outputting peripherals becomes a bottleneck. Finally, often they do not help at all, because for performance reasons, output is usually buffered and, in case of crash, the buffer is destroyed and the important information is lost, possibly resulting in starting the debugging process in the wrong place.

In some (very few) circumstances cout debugging can be appropriate, although it can always be replaced by other techniques. For these cases, here are some tips. To begin with, output must be produced on the standard error, because this channel is unbuffered and it is less likely to miss the last information before a crash. Then, printing statements should not be used directly: a macro should be defined around them (as illustrated in listing 2, adapted from [9]) so to switch debugging code on and off easily. Finally, debugging levels should be used to manage the amount of debugging information. More on debugging levels can be found in par. 2.4.1.

Listing 2: An example of cout technique - Declaration

```
#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 [9], p. 174 */
#endif
#endif /* DEBUG_H */
```

Listing 3: An example of cout technique - Implementation

```
#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 is defined to control how much debugging is outputted. Then `pmesg(100, "Foo is %l\n", foo)` can be employed to print the value of foo in case `msglevel` is set to 100 or more. Note that all this debugging code from the executable can be removed by adding `-DNDEBUG` to the preprocessor flags (sec. 3.2): 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 the file assert.h (sec. 2.5).

2.4 Logging

Logging takes the concept of printing messages, expressed in the previous paragraph, one step further. Logging is a common aid to debugging. Everyone who has tried at least once to solve some system-related problems (e.g. at machine start-up) knows how useful a log file can be. Logging means automatically recording information messages or events in order to monitor the status of your program and to diagnose problems. It is heavily used by daemons and services, exactly because their failure can affect the correct operation of the whole system. Logging is a real solution to the cout technique. It can even form the basis of software auditing, that is the evaluation of the product to ascertain its reliability.

A great example of a logging service is the GNU/Linux *syslog* program, provided by every distribution. Studying the way syslog works provides a powerful example, not to mention the expertise to solve problems with kernel, daemons and subsystems (like mail, news and web servers).

2.4.1 log4cpp C++ logging

A way of setting up logging service in a C++ program is to employ a library called *log for C++* (*log4cpp* for short). log4cpp is a library of C++ classes for flexible logging to files, syslog and other destinations. According to the authors, it is modeled after the log4j Java library, staying as close to their API as is reasonable. For a further discussion about log4cpp, see the article [10]. A discussion of the tool is reported in [11]. This paragraph presents a brief overview about this small but effective package.

log4cpp has 3 main components:

- Layouts
- Appenders
- Categories

A *Layout* class controls the appearance of the output messages. log4cpp provides the user with some predefined Layout classes, but the programmer may derive his own classes from the basic class Layout to specify any style of output message wanted.

An *Appender* class writes the trace message out to some device. The messages have been formatted by a Layout object. Again, log4cpp provides the user with some standard classes to post messages to standard output, a named file or a string buffer. The Appender class works closely with the Layout class, and once again a personalised Appender class can be derived in order to log to a different channel: for example a socket, a shared memory buffer or some sort of delayed write device.

A *Category* class does the actual logging. The two main parts of a Category are its Appenders and its priority. Priority controls which messages can be logged by a particular class. When a Category object is created, it begins with a default Appender to standard output and a default priority of none. One or more Appenders can be added to the list of destinations for logging.

The priority of a Category can be set to

1. NOTSET
2. DEBUG
3. INFO
4. NOTICE
5. WARN
6. ERROR
7. CRIT
8. ALERT
9. FATAL / EMERG

in ascending order of importance level. FATAL and EMERG are two names for the same highest level of importance. Each message is logged to a Category object. The Category object has a priority level. The message itself also has a priority level as it wends its way to the log. If the priority of the message is greater than, or equal to, the priority of the Category, then logging takes place, otherwise the message is ignored. NOTSET is the lowest and if a Category object is left with a NOTSET priority, it will accept and log any message.

Messages can be given any of these priorities except NOTSET. Therefore if a Category has been set to level WARN, then messages with levels DEBUG, INFO and NOTICE will not be logged. Messages set to WARN, ERROR, CRIT, ALERT, FATAL or EMERG will be logged.

2.4.2 log4cpp example

Here is a practical example to illustrate the usage of log4cpp. There are six initial steps to using a log4cpp log:

1. Instantiating an Appender object that will append to a log file
   `log4cpp::Appender* app =`
   `new log4cpp::FileAppender("FileAppender","/logs/testlog4cpp.log");`
2. Instantiating a Layout object
   `log4cpp::Layout* layout = new log4cpp::BasicLayout();`
3. Attaching the Layout object to the Appender
   `app->setLayout(layout);`
4. Instantiating a Category object by calling the static function
   `log4cpp::Category main_cat = log4cpp::Category::getInstance("main_cat");`
5. Attaching the Appender object to the Category as an additional Appender (in addition to the default standard out Appender), or set *additivity* (see [10]) to false first and install the Appender as the one and only Appender for that Category
   `main_cat.setAppender(app);`
6. Setting a priority for the Category
   `main_cat.setPriority(log4cpp::Priority::INFO);`

A small test program can be written to see the effect of the previous instructions. For example, including the following statements in the program

```
main_cat.info("This is some info");
main_cat.debug("This debug message will fail to write");
main_cat.alert("All hands abandon ship");
main_cat.log(log4cpp::Priority::WARN, "This will be a logged warning");
main_cat.log(priority,"Importance depends on context");
```

the following result is obtained

```
995871335 INFO main_cat : This is some info
995871335 PANIC main_cat : All hands abandon ship
995871335 WARN main_cat : This will be a logged warning
995871335 ALERT main_cat : Importance depends on context
```

As it can be seen, a message can be logged by using the member function log() with a priority. The message would not be logged if its priority is lower than the priority of the Category. The `debug` message is not recorded because the Category priority is set to INFO. Other examples can be found in the cited paper.

2.5 Defensive programming and assertions

Assertions are expressions which should evaluate to be true at a specific point in the code. If an assertion fails, a problem was found. The problem could possibly be in the assertion, but more likely it will be in the code. The important point to remember about assertions is that it make no sense to execute a program after an assertion fails

Writing assertions in the code makes assumptions explicit. In C/C++ the header file assert.h must be included and the expression you want to assert must be written as macro argument, for example `assert(var > 0)` The program will be aborted when an assertion fails, and the failure reported by a message stating the exact line of code and the exact file of the assertion.

Since assert is a macro, it can be easily removed from the final version of your code by compiling it out. If you use gcc, you must use the preprocessor flag `-DNDEBUG`.

2.6 ACI debugging technique

This paragraph could be misunderstood for a joke. On the contrary it is serious and I invite the reader not to underestimate the power of this technique with a funny name. ACI is the acronym of *Automobile Club d'Italia*, an Italian organisation that helps with car troubles. If a car gets stuck while driving on the motorway, all the driver has to do is to call ACI and a helpful mechanic will quickly come to drag him out of troubles. What does all this have to do with a debugging technique?

When a programmer is in big troubles, and does not really see a way out, he should remember the following golden rule: the best way to learn something is to teach it. This very simple principle is the key of the ACI technique. In ACI debugging the programmer must find a bystander and explain to him how the code works. Believe it or not, it is a successful technique, because calling a "mechanic" and illustrating the problem to him forces the programmer to rethink his assumptions and explain what it is really happening inside his code. Finally this technique may be employed as a form of peer review. For automatic, in contrast to human, peer review, you can read [12].

2.7 Reading the code through

This technique is quite similar to the ACI technique, with the exception that it doesn't rely on a bystander. The recipe is quite simple as well. When the programmer finds himself in complete darkness and has not the slightest idea of what is going wrong, he must print his code, leave his terminal and go to the cafeteria. After choosing his favourite drink, possibly with caffeine and sugar, he must read his code over and annotate it carefully. Understanding what a program does without actually running it is a valuable skill that a programmer must develop.

2.8 The debugger

When every other checking tool fails to detect the problem, then it is debugger's turn. A debugger allows working through the code line-by-line to find out what it is going wrong, where and why. It allows working interactively, controlling the execution of the program, stopping it at various times, inspecting variables, changing code flow whilst running.

In order to make use of a debugger, a program must be compiled with debugging information inserted. This information is provided by *debugging symbols* included by the compiler in the binaries. Debugging symbols describe where functions and variables are stored in memory. An executable with debugging symbols can run as a normal program, it is just slightly slower.

An important feature of debuggers is the possibility to set *breakpoints*. Breakpoints stop program execution on demand: the program runs normally until it is about to execute the piece of code at the same address of the breakpoint. At that point it drops back into the debugger to look at variables, or continue stepping through the code. Breakpoints are fundamental in interactive debugging, and accordingly have many options associated with them. They can be set up on a specific line number, at the beginning of a function, at a specific address, or conditionally (i.e. as soon as a condition is verified).

After stopping the program as a consequence of a breakpoint, a debugger can resume its execution. There are several ways in which this can be done. The debugger can execute just the next program line *stepping over* any function calls in the line. This way any call will be executed in one go, as if it were a single instruction. Alternatively, the debugger can *step into* a function call, executing also its code line by line. Obviously, the debugger can also go on running your program without performing any actions.

Another important feature that all decent debuggers must offer is the possibility to set *watchpoints*. Watchpoints are particular type of breakpoints which stop the code whenever a variable changes, even if the line doesn't reference the variable explicitly by name. Instead, a watchpoint looks at the memory address of the variable and alerts the programmer when something is written to it.

In large programs, adding breakpoints for every iteration of a loop is prohibitive. It is not necessary to step through each one in turn: a technique known as *binary split* can greatly simplify the debugging process. The technique consists in placing a breakpoint at the last line of the first half of the code, and running the code. If the problem does not manifest itself, then the fault is likely to be within the second half. From here, the procedure is repeated with the region where the problem is supposed to be, reducing the area under test at each iteration. At the end, the method either leaves you with just one line, or a sufficiently small routine that can be stepped through. A binary split [13] can limit the search area of a 1000 line program to just 10 steps!

Algorithm implementation errors are reasonably easy to track down with a debugger. Stepping through, looking for an invalid state or bad data, is enough: the last statement to execute either is itself wrong or at least points you at the problem. For more information, discussion and examples of how to use a debugger (and in particular *GDB*, the GNU debugger), see [14], [15], [16].

3 C/C++ related problems and solutions

In the last part of this lecture, the focus is on problems arising when programming with the C or C++ languages. C++ in now the most commonly used language in high-energy physics, therefore it seems quite appropriate to devote a large part of this review on debugging techniques to some common problems that inexperienced programmers will have to deal with from day one of their activity.

3.1 C/C++ build process

Before examining the most common problems generated by C/C++ programming, it is useful to recap the steps involved in building and running a C/C++ program. C/C++ programs can be built incrementally, i.e. the building process can be split in smaller steps. In a Unix environment, building is usually composed of 5 steps:

Preprocessing During the preprocessing phase inclusions of header files are processed and macros are expanded; the output of preprocessing is still pure C/C++ code.

Compiling Compilation is the translation of pure C/C++ code into assembly language.

Assembling The assembly code is translated into binary object code. The result is usually a file with a .o extension.

Linking Linker's task is to combines a number of object files and libraries to produce executables or libraries.

Dynamic Loading This last step consists of loading libraries (or library parts) required by a dynamically linked executable prior to actually running it.

3.2 Preprocessing

The C/C++ *preprocessor* is the program that expands macros, declares dependencies and drives conditional compilation. All the preprocessor operations are performed at textual level. This can make tracking down missing declarations difficult. It could also lead to semantic problems. If a preprocessing problem is suspected, the preprocessor should be allowed to expand the file for examination. Since the output of the preprocessor is just pure source code, debug can be done without any special tool: an editor is enough!

A real example in the Unix domain is provided by the gcc compiler with the option `-E`. This option makes gcc stop after the preprocessing stage without running the compiler. The output is preprocessed source code, which is sent to the standard output. The output can be redirected to a file.

3.3 Dynamic storage allocation

In C/C++ the programmer can explicitly allocate and deallocate dynamic storage (through malloc/free or new/delete). If memory is (de)allocated incorrectly, it can cause problems at run time (e. g. memory corruption, memory leaks).

Common errors are: trying to use memory that has not been allocated yet; accessing memory already deallocated; deallocating memory twice.

The best that can happen with a memory problem is a program crash! If the program does not crash, its behaviour becomes unpredictable, because memory corruption is like a trap: the program could run normally, pretending everything is fine, as long as it does not fall on a corrupted memory region. Sometimes memory problems do not even appear during many runs, and then, suddenly, the program crashes.

Fortunately, there are some tools to check if the program has memory problems. They can be classified into two categories:

– external libraries to be included and/or linked with the executables;
– executables which control the execution of a program.

In the first category there are, for examples, the libraries Memwatch and Electric Fence, while YAMD and Valgrind falls in the second one. An interesting discussion about these tools is reported in [17]. This lecture will discuss Electric Fence and Valgrind.

3.4 Electric Fence

Electric Fence is a C library for malloc debugging which exploits the virtual memory hardware of the system to check if and when a program exceeds the borders of a malloc buffer. At the borders of such a buffer, a red zone is added. When the program enters this zone, it is terminated immediately. The library can also detect when the program tries to access memory already released.

Because Electric Fence uses the Virtual Memory hardware to detect errors, the program will be stopped at the first instruction that causes a specific buffer to be overflown. Therefore it becomes trivial to identify the instruction that caused the error with a debugger. When memory errors are fixed, it is better to recompile the program without the library.

An example of a typical buffer overflow is reported in listing 4.

Listing 4: An example of buffer overflow

```
int main(int argc, char *argv[])
{
  double *histo;
  histo = (double *)malloc(sizeof(double) *60));
  for (int i = 0; i < 100; i++)
    histo[i] = i * i;
  return 1;
}
```

In this program, an array of 60 elements is created, but the program tries to fill it with 100 elements. As exercise, it can be compiled with `gcc -g -lefence -Wall -o memerror memerror.cpp` to check how Electric Fence spots the problem.

3.5 Valgrind

Valgrind [18] is a program which controls the execution of another program. The program can thus be compiled without any special precautions. Valgrind checks every read and write operation on memory, intercepting all calls to malloc/free new/delete. Valgrind detects problems like the usage of uninitialised memory, reading from or writing to already freed memory and reading from or writing beyond the borders of allocated memory blocks.

Valgrind tracks every byte of the memory with nine status bits: one for the accessibility and the other eight for the content, if valid. As a consequence, Valgrind can detect uninitialised areas and does not report false errors on bitfield operations. Valgrind can debug almost all dynamically linked ELF x86 executables without any need for modification or recompilation.

Here are some examples of Valgrind usage.

Listing 5: An example of buffer overflow

```
int main(int argc, char *argv[])
{
  double *histo = new double[60];
  for (int i = 0; i < 100; i++)
    histo[i] = i * i;
  return 1;
}
```

In this first example, an array of 60 elements is created. The program tries to fill it with 100 elements, which obviously ends in writing outside the boundaries of the allocated memory region. If the program is compiled with the option `-g` to include debugging symbols, it can be run under valgrind with the command `valgrind --db-attach=yes --error-limit=no ./memerror`, getting an output similar to the following one, which clearly spots the problem in the program.

```
==3252== Invalid write of size 8
==3252== at 0x80483DA: main (memerror.cpp:9)
==3252== by 0x4026F9B1: __libc_start_main (in /lib/libc.so.6)
==3252== by 0x80482F0: ??? (start.S:102)
==3252== Address 0x410B2204 is 0 bytes after a block of size 480 alloc'd
==3252== at 0x4002ACB4: malloc (in /usr/lib/valgrind/vgskin_memcheck.so)
==3252== by 0x80483A8: main (memerror.cpp:7)
==3252== by 0x4026F9B1: __libc_start_main (in /lib/libc.so.6)
==3252== by 0x80482F0: ??? (start.S:102)
==3252==
==3252== Attach to GDB ? [Return/N/n/Y/y/C/c]
```

The second example illustrates a typical error in initialisation

Listing 6: Forgotten initialisation

```
#include<iostream>
#include<math.h>
int main(int argc, char *argv[])
{
  double k, l;
  double interval = atof(argv[1]);
  if ( interval == 0.1) { k = 3.14; }
  if ( interval == 0.2) { k = 2.71; }
  l = 5.0 * exp(k);
  std::cout << "l = " << l << "\n";
  return 1;
}
```

The error does not cause a crash. The user has to give an argument as an input. If the input value is not equal to 0.1 or 0.2, the value is not initialised. Unexpected results might come out.

If valgrind is run with `valgrind --db-attach=yes --error-limit=no ./memerror` the problem is spotted quite easily, since the first message underlines the use of an unitialised value:

```
==5181== Conditional jump or move depends on uninitialised value(s)
==5181==    at 0x4134150: __ieee754_exp (in /lib/libm-2.5.so)
==5181==    by 0x8048782: main (memerror.cpp:9)
==5245==
==5245== ---- Attach to debugger ? --- [Return/N/n/Y/y/C/c] ----
```

The third and last example shows another typical error: returning a reference to a dynamically allocated object.

Listing 7: Tracking a memory leak

```
#include <string>
using namespace std;
string &xform_string_copy(const string &input);
int main(int argc, char* argv[])
{
  std::string original("I am an automatic variable");
  string& stringref = xform_string_copy(original);
}
string& xform_string_copy(const string &input)
{
  string *xformed_p = new string("I will probably be leaked!");
  //... maybe do some processing here ...
  return *xformed_p; //Callers will almost never free this object.
}
```

The effect of a debugging session with valgrind is left as an exercise to the reader.

3.6 System call examination

A *system call tracer* is a program that allows you to examine problems at the boundary between your code and the operating system. A user program cannot interact directly with the kernel of the operating system. The program is actually executed in what is called user space. If a user space program wants to access a hard disk, for example, it cannot do it directly but it must call an appropriate system function, which is in charge of moving data between the program and the disk. To find out which function is employed by the program, a system call tracer is the right tool. The tracer shows which system calls a process makes together with the passed parameters and the returned value. Unfortunately a tracer cannot tell where a system call was made in the code, but only that the call took place. The exact place has to be reconstructed. A good idea is to employ a system call tracer together with a logger. This is a good example of how to combine different techniques to spot a problem.

3.6.1 strace, the GNU/Linux system tracer

The standard system call tracer in the GNU/Linux system is *strace*. strace is a powerful tool which shows all the system calls issued by a user-space program. strace has several advantages. It displays the arguments to the calls and returns values in symbolic (i.e. human readable) form. strace receives information from the kernel and does not require the kernel to be built in any special way; therefore it runs on every machine.

3.6.2 Example

Let's see a simple example of strace usage. This is a complete program; it can be compiled with `g++ -o straceTest strace.cpp` and run as it is. As a further exercise and application of the technique explained in 2.7, it is recommended studying the program carefully before starting playing with it on a computer, trying to understand what the problem is only by visual examination.

Listing 8: A program with a hardcoded datum

```
#include <iostream> // for I/O
#include <string> // for strings
#include <fstream> // for file I/O
#include <cstdlib> // for exit()
using namespace std;
int main (int argc, char* argv[])
{
  string filename;
  string basename;
  string extname;
  string tmpname;
  const string suffix("tmp");
  /* for each commandline
  argument (which is an ordinary Cstring) */
  for (int i=1; i<argc; ++i)
  {
    filename = argv[i]; // process argument as file name
    string::size_type idx = filename.find('.'); // search period in name
    if (idx == string::npos)
    {
      // file name does not contain any period
      tmpname = filename; // HERE IS THE ERROR
      //tmpname = filename + '.' + suffix;
    }
    else tmpname = filename;
    // print file name and temporary name
    // cout << filename << " => " << tmpname << endl; // USEFUL
  }
  ifstream file(tmpname.c_str());
  if (!file)
  {
    cerr << "Can't open input file \"" << filename << ".tmp\"\n";
    exit(EXIT_FAILURE);
  }
  char c;
  while (file.get(c))
  cout.put(c);
}
```

This simple program tries to access a text file with the (hardcoded) extension `tmp` inside the current directory. The name of the file to be opened must be given as a command line parameter. The program attaches the suffix to the name and opens the file. To perform the exercise, a text file with a suitable name (e.g. list.tmp) has to be create, using the command `ls > list.tmp` or with an editor of your choice.

Running the program will generate the error message `Can't open input file "list.tmp"`. The program cannot find the input file, but that is pretty strange, because listing the content of the directory will show the file right there. Obviously the program is affected by a bug.

As can be seen in the source code of listing 8, the programmer simply forgot to attach the extension to the name. In this program there is actually a second, logical, bug. In an attempt to communicate the failure of file opening, the programmer inserted an error message (note the usage of the unbuffered `cerr` stream). But the error message is misleading, because the programmer hardcoded the extension of the file name in the message. It would have been wiser to print out just the name of the file to be read. In this way, the problem could have been recognised immediately.

Let us see how to usefully employ strace to find the problem. Start strace with the command line `strace -o strace.out ./straceTest list`. The output of strace is written into the file `strace.out`. This is more convenient than receiving the output on screen. Scrolling down the list of system calls, the point where the program attempts to read the file is market by the system call `open`.

```
open("list", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or directory)
write(2, "Can\'t open input file \"", 23) = 23
write(2, "list", 4)                      = 4
write(2, ".tmp\"\n", 6)                  = 6
exit_group(1)                            = ?
```

Examining the list of system calls it can be seen immediately that the problem is with the name of the file, because the parameter of the open function tells us the program is trying to open the file called `list`, not `list.tmp`. but the file is not there, and an error value `-1` is returned.

Conclusion

This lecture reviewed a set of useful debugging techniques. Good debugging skills are acquired through experience, as a general theory of debugging does not exist. Therefore, the final remark for the reader is to make a lot of experiments by himself, investigating deeply the reasons of program failures. This way, the reader will gain an invaluable insight not only into all the nuisances software can cause, but especially into the way it is built and runs: a key skill to catch bugs.

Acknowledgements

I would like to thank very much J. H. M. Dassen and I. G. Sprinkhuizen-Kuyper for letting me use some of their material on debugging techniques [19]; P. F. Zema for useful technical comments and discussions on GNU/Linux debugging, E. Castorina for a critical review of the lecture slides and E. Eisenhandler for revising the manuscript in its pedagogical content. I would also like to thank F. Flückiger for providing me the opportunity to present this lecture to CERN iCSC.

References

[1] Wikipedia contributors. *Debugging [Internet]*. Wikipedia, The Free Encyclopedia; 4 June 2007, 20:01 UTC [cited March 2, 2008]. Available from: `http://en.wikipedia.org/w/index.php?title=Debugging&oldid=129632693`.

[2] T. Parr, *Learn the essential of debugging*, IBM developerWorks journal, December 2004. Available from: `http://www.ibm.com/developerworks/web/library/wa-debug.html`.

[3] A. M. Hayashi, *Rough sailing for smart ships*, Scientific American, November 1998, Vol. 279, Issue 5, p46.

[4] *Mars Climate Orbiter Mishap Investigation Board Phase I Report*, November 10, 1999. Available from: `ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO\_report.pdf`.

[5] *Prof. G. Santor's list of famous computing bugs.* Available from: `http://infotech.fanshawec.on.ca/gsantor/Computing/FamousBugs.htm`

[6] B. Jacobsen, *Tools and Techniques*, Lecture given at CERN School of Computing 2004, Vico Equense, Italy. Available from: `http://csc.web.cern.ch/csc/2004`

[7] P. Tonella, *Software Evolution and Testing*, Lecture given at CERN School of Computing 2004, Vico Equense, Italy. Available from: `http://csc.web.cern.ch/csc/2004`

[8] R. M. Stallman and the GCC Developer Community, *Using the GNU Compiler Collection*, The Free Software Foundation. Available from: `http://gcc.gnu.org/onlinedocs`

[9] B. Kernighan, D. Ritchie, *The C Programming Language*, 2nd edition, (Englewood Cliffs, NJ: Prentice Hall).

[10] M. Budlong, *Logging and Tracing in C++ Simplified*, Solaris Developer Technical Articles, August 2001. Available from: `http://developers.sun.com/solaris/articles/logging.html`

[11] J. Smith, *Using log4cpp*, Jefficus World, June 13 2004. See `http://jefficus.usask.ca`

[12] S. Goodwin, D. Wilson, *Walking Upright*, Linux Magazine, February 2003, Vol. 27, Issue 2, p76.

[13] S. Goodwin, *The Pleasure Principle*, Linux Magazine, June 2003, Vol. 31, Issue 6, p64.

[14] F. Rooms, *Some advanced techniques in C under Linux*.
Available from: `http://telin.ugent.be/~frooms/publications/cleseng.pdf`

[15] W. Mauerer, *Visual Debugging with ddd*, The Linux Gazette, December 2001, Issue 73.
Available from: `http://linuxgazette.net/issue73/index.html`

[16] R. M. Stallman et al., Debugging with GDB, The Free Software Foundation.
Available from: `http://www.gnu.org/software/gdb/documentation/`

[17] S. Best, *Mastering Linux debugging techniques*, IBM developerWorks journal, August 2002.

[18] Valgrind Developers, *Valgrind Documentation*. Available from: `http://valgrind.org`

[19] J.H.M. Dassen, I.G. Sprinkhuizen-Kuyper, *Debugging C and C++ code in a Unix environment*, Universiteit Leiden, Leiden, 1999.