Paolo Adragna
Università degli Studi di Siena

Debugging Techniques

Why Debugging?

Debugging is a **fundamental** part of *programmers' everyday activity*....

... but *some people* consider it an **annoying** option...

USS Yorktown (1998)

*A crew member of the guided-missile cruiser USS Yorktown mistakenly entered a zero for a 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 the water for several hours because a program **didn't check for valid input**.*
(reported in Scientific American, November 1998)

Mars Climate Orbiter (1999)

*The 125 million dollar Mars Climate Orbiter is assumed **lost** by officials at NASA. The failure responsible for loss of the orbiter is attributed to a failure of NASA's **system engineer 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.***
http://mars.jpl.nasa.gov/msp98/orbiter/

Lecture Programme

- Part I - General Aspects of Debugging
- Part II - General Debugging
- Part III - C/C++ Related Problems and Solvers

Part I

General Aspects of Debugging

Part One - General Aspects of Debugging

The debugging process involves:

- Localising a bug
- Classifying a bug
- Understanding a bug
- Repairing a bug

Localising a Bug

```
#include <iostream>
// A scoping example
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
}
```

"*You know what your code **should** do You notice it **does not** do that so noticing a bug is easy*", you might say...

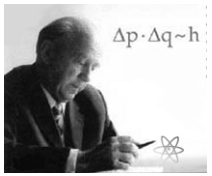
Classifying a Bug

- Since experiences with bugs have often a common background, we may attempt a *classification*:
  - **Syntactical Errors**: errors your compiler should catch.
  - **Build Errors**: errors from using object files not rebuilt after a change in some source.
  - **Basic Semantic Errors**: using uninitialized variables, dead code, type problems.
  - **Semantic Errors**: using wrong variables, exchanging operator (e. g. & instead of &&)

Classifying a Bug

A funny "physical" classification
Bohrbugs and *Heisembugs*

Bohrbugs are *deterministic*:
a particular input will always manifest them.



Heisembugs are *random*: difficult to reproduce reliably

Understanding a Bug

- Understand a bug fully before attempting to fix it
- Ask yourself some questions:
  - Have I found the source of the problem or only a symptom?
  - Have I made similar mistakes (especially wrong assumptions) elsewhere in the code?
  - Is this only a programming error or is there a more fundamental problem (e. g. incorrect algorithm)?

Repairing a Bug

- Repairing a bug is more than modifying code. Make sure you document your fix in the code and **test it properly**.
- After repair, what did you *learn* from it?
  - How did you notice the bug? This may help you writing a *test case*.
  - How did you track it down? This will give you a better insight on the approach to choose in similar circumstances.
  - What type of bug did you encounter?

Repairing a Bug

- After repair, what did you learn from it?
  - Do you encounter this bug often? If so, what could you do to prevent it from re-occurring?
  - What you have learnt is valuable: try to communicate it with your collegues
  - Unjustified assumptions?
- After repairing a bug, **write a test case** to make sure it does not happen again

Part Two

General Debugging

Part Two – General Debugging

A) Exploiting Compiler Feature

B) Reading The Right Documentation

C) The Abused *cout* Debugging Technique

D) Logging

E) Defensive Programming

F) ACI Debugging Technique

G) Walking Through The Code

H) The Debugger

Exploiting Compiler Features (General)

- A good compiler can do an amount of static analysis on your code (the analysis of those aspects that can be studied without execution)
- Static analysis can help in detecting a number of basic semantic problems (e. g. type mismatch, dead code)

Exploiting Compiler Features (gcc)

- For gcc there are a number of options that affect which static analysis can be performed
  - Wall -W
- Also recommended when writing new code
  - Wshadow
  - Wpointer-arith
  - Wcast-equal
  - Wcast-align
  - Wstrict-prototype

Exploiting Compiler Feature (gcc)

- A number of optimizations are supported. Some of these trigger gcc to do extensive code flow analysis, removing dead code.
- Recommended for normal use: -O2
- Warning: optimisation **kills** debugging, so you have to choose
  - Example: gcc -O3 **or** gcc -g -O0

Reading the Right Documentation

- Take the time to find at your fingertips relevant documentation for:
  - your task
  - your tools
  - your libraries
  - your algorithm
- You do not need to know everything
- You need to be aware what documentation is relevant and what is its purpose

The Abused *cout* Technique

- This technique is encountered too often.
- It consists of ad hoc insertion of lot of printing statement to track the control flow and data values during the execution of a piece of code
- Disadvantages
  - It is **very** ad hoc
  - It clobbers the normal output
  - Slows the program down considerably
  - Often it does not help (output buffered)

The Abused *cout* Technique

- If you consider using debugging, check out the use of assertion and of a debugger, much more effective and time saving
- In some circumstances *cout* debugging is appropriate. Some tips:
  - Produce output on standard error (unbuffered)
  - Do not use printing statements directly: define a macro around them
  - Use debugging level to manage the amount of debugging information

cout Technique - Example

```
#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 */
```


Logging

- Logging is a common aid to debugging
- Heavily used by daemon and services
- It is a **real solution** to the cout technique
- It records information messages which monitor the status of your program
- They can even form the basis of software auditing
- A sensible method is to classify log messages and label them with a priority level

log4cpp - C++ Logging

Log4cpp has 3 main components:

- Categories
- Appenders
- Layouts

A **layout** class controls what the output message is going to look like.

You may derive your own classes from Layout or use the provided SimpleLayout and BasicLayout

log4cpp - C++ Logging

An **appender** class writes the trace message, formatted by a layout object, out to some device

log4cpp comes with classes to append to *standard output*, a *named file*, or a *string buffer:*

- FileAppender
- OstreamAppender
- StringQueueAppender

Once again you may derive your **own** appender (e.g. to a socket, a shared memory buffer...)

log4cpp - C++ Logging

A **category** class does the actual logging.

The two main parts of a category are its **appenders** and its **priority**

The priority of a category can be set to:

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

in ascending order of importance level

log4cpp - C++ Logging

Each message is logged to a **category** object

The category object has a **priority level**

Priority controls which messages can be logged by a particular class.

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

Log4cpp - Example

There are **six initial steps** to using a log4cpp log:

- Instantiate an appender object that will append to a log file

```
log4cpp::Appender* app = new log4cpp::FileAppender
        ("FileAppender","/logs/testlog4cpp.log");
```

- Instantiate a layout object

```
log4cpp::Layout* layout = new log4cpp::BasicLayout();
```

- Attach the layout object to the appender

```
app->setLayout(layout);
```

Log4cpp - Example

- Instantiate a category object by calling the static function

  log4cpp::Layout layout = new log4cpp::BasicLayout();*

- Attach the appender object to the category as an additional appender (in addition to the default standard out appender), or set Additivity to false first and install the appender as the one and only appender for that category

  main_cat.setAppender(app);

- Set a priority for the category

  main_cat.setPriority(log4cpp::Priority::INFO);

Log4cpp - Example

Some examples:

```
main_cat.info("This is some info");
main_cat.debug("This debug message will fail to write");
main_cat.alert("All hands abandon ship");

/* you can log by using a log() method with a priority */
main_cat.log(log4cpp::Priority::WARN, "This will be a logged
warning");

/* this would not be logged if priority == DEBUG, because the
category priority is set to INFO */
main_cat.log(priority,"Importance depends on context");
```

Other example in the cited paper (see Bibliography)

Log4cpp – Logfile Example

A tipical logfile:

```
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
995871335 ERROR main_cat : And this will be an error
995871335 INFO main_cat : info
995871335 NOTICE main_cat : notice
995871335 WARN main_cat : warn
```


Defensive Programming and the assert Macro

- Take a look at your code: in every part you make a lot of assumptions about other parts
- Assertions are expressions you should evaluate to be true at a specific point in your code
- If an assertion fails, you have found a problem (possibly in the assertion, more likely in the code)
- It make no sense to execute after an assertion fails

Defensive Programming and the assert Macro

- Writing assertions makes your assumptions explicit
- In C/C++ you can #include <assert.h> and write the expression you want to assert as macro argument
- With assert macros your program will be aborted when an assertion fails
- An assertion failure is reported by a message

ACI Debugging Technique

ACI, *only a joke*...

- The technique name derive from Automobile Club d'Italia, an Italian organisation that helps with car troubles...

ACI Debugging Technique

ACI, *not only a joke*...

- Based on a simple principle: the best way to learn thing is to **teach** them

In ACI debugging you find a *bystander* and explain to her how your code works

This forces you to **rethink** your assumption and **explain** what is really happening
It can be a form of *peer review*

Walking through the Code

This technique is similar to the ACI technique.

The recipe:

- Print your code
- Leave your terminal
- Go to cafeteria
- Take the beverage of your choice, if possible with caffeine and sugar
- Read your code and annotate it carefully

The Debugger

- When every other checking tool fails detecting the problem, then it is debugger's turn.
- A debugger allows to work through the code line-by-line to find out where and why it is going wrong.
- You can interactively control the program run, stop it at various times, inspect variables, change code flow whilst running.

The Debugger

- In order to make use of a debugger, a program must be compiled with debugging information inserted (*debugging symbols*)
- **Debugging symbols** describe where the function and variables are stored in memory
- An executables with debugging symbols can run as a normal program, even if slightly slower

Breakpoints

- **Breakpoints** stop a program when needed
  - The program runs normally until it is about to execute the piece of code at the same address of the breakpoint
  - at that point, the program drops back into the debugger and we can look at variables, or continue stepping through the code.
- Breakpoints are fundamental in **interactive debugging**

Breakpoints

- Breakpoints have many options. They can be set up:
  - on a specific line number
  - at the beginning of a function
  - at a specific address
  - conditionally

Debugging Commands

After stopping (e.g. at a breakpoint) every debugger can:

- execute next program line stepping over any function calls in the line
- execute next program line stepping into any function calls in the line
- continuing running your program

Watchpoints

- **Watchpoints** are a particular type of breakpoints
- A watchpoint stops the code whenever a variable changes, even if the line doesn't reference the variable explicitly by name
- A watchpoint looks at the memory address of the variable and alerts you when something is written to it

Binary Split

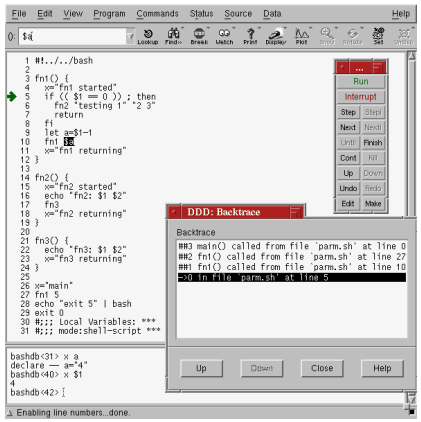
- In large programs, adding breakpoints for every iteration of the loop is prohibitive
- It is not necessary to step through each one in turn, but employ a technique known as binary split:
  - We place a breakpoint after the first of the code and run it.
  - If the problem has not showed up, then it is likely to be a fault with the last half.

Binary Split

  - From here, we can ask the question again, reducing the area under test to the first or the second quarter
  - This question can be asked repeatedly until we're down to just one line, or sufficiently small routine that we can step through line-by-line

A binary split can limit the search area of a 1000 line program to just 10 steps!

DDD – GUI for gdb



Data
Display
Debugger

Powerful interface to gdb with extra features

Try it on our first example

Part III

C/C++ Related Problems and Solvers

Part Three – C/C++ Related Problems and Solvers

A) Preprocessor

B) Dynamic Storage Allocation

C) System Call Examination

C/C++ Build Process

A brief review of steps involved in building and running a program

- **Preprocessing** – header files, inclusion and macro processing; output in pure C/C++ code
- **Compiling** – translation of pure C/C++ code to assembly language
- **Assembling** – translation of assembly code into binary object code

C/C++ Build Process

- **Linking** – linker combines a number of object files and libraries to produce executables or libraries
- **Dynamic Loading** - libraries (or library parts) required by a dynamically linked executables are loaded prior to actual running the executables

Preprocessor

- The C/C++ preprocessor:
  - expands macros
  - declares dependencies
  - drives conditional compilation
- Preprocessor operations are performed at *textual* level. This can make tracking down missing declaration difficult or lead to semantic problem

Preprocessor

- If you suspect a preprocessing problem, let the preproccessor expand the file for examination
- Example: gcc -E
  - Stops after the preprocessing stage without running the compiler. The output is preprocessed source code, which is sent to the standard output

Dynamic Storage Allocation

- In C/C++ you have to 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 leak)
- Common errors are: trying to use memory that has not been allocated yet; to access memory already deallocated; deallocating memory twice

Memory Allocation Debugging Tools

When you have a memory problem, the **best** it can happen is a program crash!!!

Basically two categories of tools:

- External libraries to be included and/or linked
  - MEMWATCH
  - Electric Fence
- Executables which controls program's run
  - YAMD
  - Valgrind

Electric Fence

- Electric Fence is C library for *malloc* debugging
- It 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 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.

Electric Fence

- Because Electric Fence uses the Virtual Memory hardware to detect errors, the program will be stopped at the first instruction that causes a certain buffer to be exceeded.
- 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.

Example – Memory Error

An array of 60 elements is created.
The program tries to fill it with 100 elements

```
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;
}
```

Compile the program with:

```
g++ -g -lefence -Wall -o memerror memerror.cpp
```

Valgrind

- Valgrind checks every reading and writing operation on memory, intercepting all calls to malloc/free new/delete
- Valgrind can detect problems like:
  - usage of uninitialised memory
  - reading from / writing to freed memory
  - reading from / writing beyond the borders of allocated blocks

Valgrind

- 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 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.

Example – Memory Error

An array of 60 elements is created.
The program tries to fill it with 100 elements

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

Compile the program with:
g++ -g -Wall -o memerror memerror.cpp

Example – Memory Error

```
valgrind --gdb-attach=yes --error-limit=no ./memerror
.....
==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] ----
```

Example – Forgetting the Initialisation

Consider the following simple program

```
#include<iostream>
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;
}
```

- Compile with:

g++ -lm -g -o val3 initia1.cpp

- The error doesn't 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 initialized
- We may get unexpected results

Example – Forgetting the Initialisation

valgrind --gdb-attach=yes --error-limit=no --leak-check=yes memerror

```
==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== ---- Attach to GDB ? --- [Return/N/n/Y/y/C/c] ----
```


Example – Tracking Memory Leak

Typical Error

Returning a Reference to a Dynamically Allocated Object

```
#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.
}
```


System Call Examination

- A **System Call Tracer** allows you to examine problems at the boundary between your code and operating system
- The tracer shows what system calls a process makes (with parameters and return value)
- A tracer cannot tell you *where* a system call was made in your code.
- The *exact* place has to be reconstructed

strace, the Linux System Tracer

- ***strace*** is a powerful tool which shows all the system calls issued by a user-space program.
- *strace* displays the arguments to the calls and returns values in symbolic form.
- *strace* receives information from the kernel and does not require the kernel to be built in any special way.

strace example

```
#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");
```


strace example

```
/* for each command-line argument (which is an ordinary C-string)*/
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
    }
```

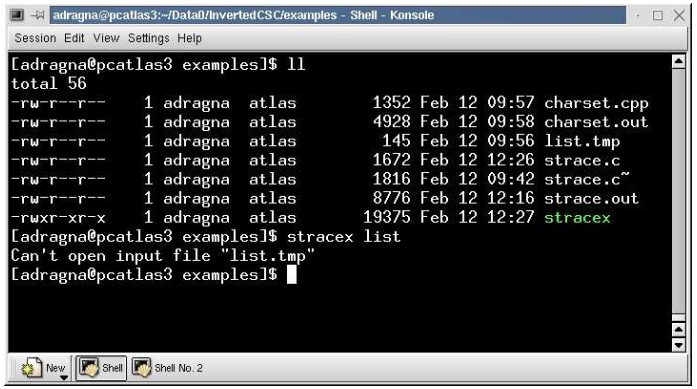

strace example

```
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);
}
```

- Create a simple text file and run the program.
- The program won't find the input file...

strace example

... but there it is!



strace example

Let's start *strace*: **`strace -o strace.out stracex list`**

```
brk(0x804a76c)          = 0x804a76c
brk(0x804b000)          = 0x804b000
open("list", O_RDONLY)  = -1 ENOENT (No such file or directory)
write(2, "C", 1)        = 1
write(2, "a", 1)        = 1
write(2, "n", 1)        = 1
write(2, "\'", 1)       = 1
write(2, "t", 1)        = 1
write(2, " ", 1)        = 1
write(2, "o", 1)        = 1
write(2, "p", 1)        = 1
write(2, "e", 1)        = 1
write(2, "n", 1)        = 1
```


Acknowledgments

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

A big thank also to P. F. Zema, my collegue in ATLAS, for useful technical comments and ideas exchange on Linux debugging.

Thanks to E. Castorina for a critical review of the lecture slides

Bibliography

- For more famous bugs, take a look to Prof. G Santor's site: http://infotech.fanshawec.on.ca/gsantor/Computing/FamousBugs.htm
- J.H.M. Dassen, I.G. Sprinkhuizen-Kuyper, *Debugging C and C++ code in a Unix environment*, Universiteit Leiden, Leiden, 1999
- T. Parr, *Learn the essential of debugging*, IBM developerWorks journal, Dec 2004
- S. Best, *Mastering Linux debugging techniques,* IBM developerWorks journal, Aug 2002
- S. Goodwin, *The Pleasure Principle*, Linux Magazine 31 (2003) 64 - 69

Bibliography

- *gdb User Manual*
- *gcc User Manual*
- *Valgrind User Manual*
- F. Rooms, *Some advanced techniques in C under Linux*
- W. Mauerer, *Visual Debugging with ddd*, The Linux Gazette, Jan 2001
- M. Budlong, *Logging and Tracing in C++ Simplified*, Sun Developers Technical Articles, 2001
- S. Goodwin, D. Wilson, *Walking Upright,* Linux Magazine 27 (2003) 76 - 80
- J. World, *Using Log4c,* online at http://jefficus.usask.ca

Backup Slides

Localising a Bug

- "You know what your code should do, you notice it does not do that so noticing a bug is easy", you might say...
- Noticing a bug implies testing, so this easiness is completely deceptive
- In case of a test failure you have to see what went wrong, so prepare your tests carefully

Introduction

- When your program contains a bug, it is likely that, somewhere in the code, a condition you believe to be true is actually false
- Finding your bug is a process of confirming what you believe is true until you find something that is false.
- "My program doesn't work" is not an acceptable statement

Introduction

- The importance of the way how to find errors and fix them in the life cycle of a software product is a task whose importance cannot be stressed enough over and over
- Finding errors is not just an unavoidable part in the development cycle but vital part of every software system's lifespan.