

# Programming for Chemical and Life Science Informatics

I573 - Week 2  
(Debugging & Profiling)

Rajarshi Guha

20<sup>th</sup> January, 2009

# Outline

- 1 Debugging
- 2 Profiling
- 3 Static Analysis

# Bugs

- They **will** happen - doesn't matter how good you are!
- Good programming practise can help reduce bugs
- Also helps to catch bugs before they become too serious
- Inevitably you will have to fix them
  - ▶ Identify the bug
  - ▶ Fix it
  - ▶ Ensure it doesn't happen again
- [Why Programs Fail - A Guide to Systematic Debugging](#) by Andreas Zeller

# Types of Bugs - Compile Time

- Can be caught by the compiler
- The most common form of bug is the syntax error
  - ▶ misspelled keywords
  - ▶ bad indentation or missing semi colons
  - ▶ wrong type assignments - `int x = ‘‘Hello’’;`
- Traditionally the compiler catches these
- Nowadays, the editor can catch these before actually building the project
- Depending on the language, the editor will catch some subtle problems in the code
  - ▶ Usually seen in Java or C# IDE's

# Warnings - Are They Useful?

- Depending on the language / editor, you may also get a list of warnings
- These won't stop your program working
- Might indicate possible problems later on
- If you're using gcc use the `-Wall` option and heed them
- For IDE's (Eclipse, IDEA etc.) consider the suggestions
  - ▶ If you're using Java 5, think about adding type information to `Collection` objects
  - ▶ Avoid unnecessary casting, make use of autoboxing

## Warnings - Are They Useful?

- Adding type information to collections can be useful when reading code
- Allows the compiler to catch errors

```
List tokens = new ArrayList();  
for (int i = 0; i < 10; i++)  
    tokens.add("Hello");
```

versus

```
List<String> tokens = new ArrayList<String>();  
for (int i = 0; i < 10; i++)  
    tokens.add("Hello");
```

# Types of Bugs - Run Time

- These appear when running the program
- Symptoms
  - ▶ program crashes
  - ▶ wrong or absurd results
- Can also be intermittent - very frustrating!
  - ▶ a.k.a Hiesenbug
- Static code analysis can help prevent these bugs
  - ▶ Essentially makes run time bugs into compile time bugs
  - ▶ An active research topic

# Debugging Techniques

- Ranges from simple to sophisticated
- Simplest is *print* debugging
  - ▶ Insert `print` or `printf` statements
  - ▶ Allows you to see the state of variables
- This approach allows you to crudely identify where in the code the problem is occurring
- Very tedious, clutters your code
- Use the proper tools - the debugger

# Debugging Tools

- **GDB** is the most common CLI debugger found on Linux systems. It handles many languages supported by GCC (such as C, C++, Fortran, Pascal, Objective-C)
- GUI frontends are available
  - ▶ **GUD** - useful for debugging from within Emacs
  - ▶ **DDD**
- **valgrind** is not a debugger *per se*, but allows you track memory usage and bugs related to memory (allocation, deallocation, out-of-bounds). It's a suite of program that provide allow thread debugging, heap and cache profiling etc.
- **PDB** is a Python debugger built into the standard library
- **Firebug** is a debugging IDE for web applications (i.e., CSS, HTML, Javascript, AJAX etc.) and is part of Mozilla Firefox

# Debugging

## Preparing for Debugging

- Depends on the language
- For C or C++ programs with gcc supply the `-g` command line option
- If you're working in Eclipse, IDEA etc, just run in debug mode

## Terminology

- **breakpoint** - a point in the source code where program execution will stop
- **watch** - watches the value of a variable as the program executes
- **step** - execute a single line of source code

# Debugging Example

```
#include <stdio.h>
/* Print the sum of all integers up to and including the square root of the user specified value */
int main(Char **argv, int argc) {
    int i; int n = 0; int sum = 0;
    printf("Enter a number: ");
    scanf("%d", n);
    for (i = 0; i < n; i++) {
        sum += i;
    }
    printf("sum = %d\n", sum);
    return (0);
}
```

Based on Norman Matloff's excellent [gdb tutorial](#)

# Debugging Example - Round 1

## Compile

```
[rguha@cheminfo ~]$ gcc -ggdb -o bug bug.c  
bug.c:6: error: expected ')' before '*' token
```

## Fix

- Syntax error on line 6
- Misspelled char
- A good editor would have caught this before compiling

## Debugging Example - Round 2

### Compile

```
[rguha@cheminfo ~]$ gcc -ggdb -o bug bug.c
```

### Run

```
[rguha@cheminfo ~]$ ./bug  
Enter a number: 9  
Segmentation fault
```

Time to fire up the debugger

## Debugging Example - Round 2

### Compile

```
[rguha@cheminfo ~]$ gcc -ggdb -o bug bug.c
```

### Run

```
[rguha@cheminfo ~]$ ./bug  
Enter a number: 9  
Segmentation fault
```

**Time to fire up the debugger**

## Debugging Example - Round 3

```
[rguha@cheminfo ~]$ gdb bug
GNU gdb Red Hat Linux (6.5-25.el5rh)
Copyright (C) 2006 Free Software Foundation, Inc.
```

```
[snip]
```

```
(gdb) run
Starting program: /home/rguha/bug
Enter a number: 9
```

```
Program received signal SIGSEGV, Segmentation fault.
0x003af371 in _IO_vfscanf_internal () from /lib/libc.so.6
```

## Debugging Example - Round 3

```
(gdb) bt
#0  0x003af371 in _IO_vfscanf_internal () from /lib/libc.so.6
#1  0x003b45fb in scanf () from /lib/libc.so.6
#2  0x0804841b in main () at bug.c:13
```

- `bt` is short for backtrace
- Read from top to bottom
- Indicates that `scanf` is the problem and is located at line 13 in `bug.c`
- But `scanf` is a library function, so should be OK
- Are we making a mistake in calling it?

## Debugging Example - Round 3

```
(gdb) l 13
5      including the square root of the user specified
6      value
7      */
8      int main(char **argv, int argc) {
9          int i;
10         int n = 0;
11         int sum = 0;
12         printf("Enter a number: ");
13         scanf("%d", n);
14         for (i = 0; i < n; i++) {
```

- l lists the specified line and a few lines before it and after
- It looks like our call to scanf is wrong

## Debugging Example - Round 4

### Compile

```
[rguha@cheminfo ~]$ gcc -ggdb -o bug bug.c
```

### Run

```
[rguha@cheminfo ~]$ ./bug  
Enter a number: 9  
sum = 36
```

- Runs with no error
- Gives the wrong result. The answer should be 6, not 36
- Back to the debugger

## Debugging Example - Round 4

```
(gdb) break 14
```

```
Breakpoint 1 at 0x80483f2: file bug.c, line 14.
```

```
(gdb) run
```

```
Starting program: /home/rguha/bug
```

```
Enter a number: 9
```

```
Breakpoint 1, main () at bug.c:14
```

```
14         for (i = 0; i < n; i++) {
```

```
(gdb) n
```

```
15             sum += i;
```

```
(gdb) p i
```

```
$1 = 0
```

- Set a breakpoint at line 14 - the code will stop there
- Execute the statement using the n (next) command
- View the value of i using the p (print) command

## Debugging Example - Round 4

```
(gdb) break 14
Breakpoint 1 at 0x80483f2: file bug.c, line 14.
(gdb) n 7
14      for (i = 0; i < n; i++) {
(gdb) p i
$12 3
...
(gdb) p n
$15 9
```

- `n 7` means to execute the next 7 statements
- This will run through the loop 3 more times
- The value of `i` should be 3
- If we step through again, we see that `i` becomes 4.
- Therefore `n` is not really the square root of 9

# Debugging Example - Round 5

## Fix

```
for (i = 0; i < n; i++) {
```

becomes

```
for (i = 0; i <= (int) sqrt(n); i++) {
```

## Compile & Run

```
[rguha@cheminfo ~]$ gcc -ggdb -o bug bug.c -lm
```

```
[rguha@cheminfo ~]$ ./bug
```

```
Enter a number: 9
```

```
sum = 6
```

It works!

# Debugging Example - Round 5

## Fix

```
for (i = 0; i < n; i++) {
```

becomes

```
for (i = 0; i <= (int) sqrt(n); i++) {
```

## Compile & Run

```
[rguha@cheminfo ~]$ gcc -ggdb -o bug bug.c -lm
```

```
[rguha@cheminfo ~]$ ./bug
```

```
Enter a number: 9
```

```
sum = 6
```

It works!

# Debugging

- We looked at some simple (contrived) bugs
- Memory (de)allocation is a common source of errors in C and C++
- Python, Java etc generally avoid these sort of problems by having their own garbage collection schemes
- If you work in C++, consider Boost's `smart_ptr`
- `valgrind` is a highly recommended tool
  - ▶ Allocation/deallocation problems
  - ▶ Illegal read/writes
  - ▶ Uninitialized values
  - ▶ Linux only

# Memory Bugs

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

int main(int argc, char **argv) {
    char *text;
    int i = 0;

    for (i = 0; i < 1000; i++) {
        text = (char*) malloc(sizeof(char) * i * 1000);
        printf("malloc'ed %d bytes\n", sizeof(char)*i*1000);
    }
}
```

- What's the problem?
- How do we find it?
- How do we find it efficiently?

# Memory Bugs - Valgrind

```
[rguha@localhost tmp]$ valgrind ./a.out
...
...
==9337== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 14 from 1)
==9337== malloc/free: in use at exit: 49,950,000 bytes in 1,000 blocks.
==9337== malloc/free: 1,000 allocs, 0 frees, 49,950,000 bytes allocated.
==9337== For counts of detected errors, rerun with: -v
==9337== searching for pointers to 1,000 not-freed blocks.
==9337== checked 47,564 bytes.
==9337==
==9337== LEAK SUMMARY:
==9337==    definitely lost: 48,920,300 bytes in 980 blocks.
==9337==    possibly lost: 1,029,700 bytes in 20 blocks.
==9337==    still reachable: 0 bytes in 0 blocks.
==9337==    suppressed: 0 bytes in 0 blocks.
==9337== Use --leak-check=full to see details of leaked memory.
```

# Memory Bugs - Valgrind

```
[rguha@localhost tmp]$ valgrind ./a.out
...
...
==9496== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 1)
==9496== malloc/free: in use at exit: 0 bytes in 0 blocks.
==9496== malloc/free: 1,000 allocs, 1,000 frees, 49,950,000 bytes allocated
==9496== For counts of detected errors, rerun with: -v
==9496== All heap blocks were freed -- no leaks are possible.
```

# Memory Bugs - Valgrind

- That was a trivial case, easily caught by eye
- Valgrind shines when dealing with larger projects
- Can generate *copious* output, so GUI's can be useful.
- A slightly non-trivial case might be to check whether OpenBabel 2.2.0b1 leaks memory when converting file formats

# Memory Bugs - Valgrind

```
[rguha@cheminfo tmp]$ echo 'c1cccc1CC#N' | valgrind babel -ismi -oinchi
...
...
==9373== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 163 from 1)
==9373== malloc/free: in use at exit: 70,707 bytes in 559 blocks.
==9373== malloc/free: 7,803 allocs, 7,244 frees, 747,064 bytes allocated.
==9373== For counts of detected errors, rerun with: -v
==9373== searching for pointers to 559 not-freed blocks.
==9373== checked 763,980 bytes.
==9373==
==9373== LEAK SUMMARY:
==9373==    definitely lost: 0 bytes in 0 blocks.
==9373==    possibly lost: 1,236 bytes in 65 blocks.
==9373==    still reachable: 69,471 bytes in 494 blocks.
==9373==    suppressed: 0 bytes in 0 blocks.
==9373== Reachable blocks (those to which a pointer was found) are not shown.
==9373== To see them, rerun with: --show-reachable=yes
```

# Memory Bugs

- Analysing OpenBabel is a little tricky
- Each file format is in the form of a dynamic library (.so file)
- So memory leaks could occur in OpenBabel or one of the dynamic libs
- Should use the `--leak-check-full=yes` option - will show details in offending libs
- If you write medium to big C/C++ programs, **use** valgrind

# Debugging

- Integrating your debugger with the IDE is nice
  - GUI interface, click on the line you want to break at
  - Perform operations on variables
  - Generally makes debugging a little more fun

The screenshot illustrates the integration of a debugger with an IDE. It is divided into three main sections:

- Code Editor:** Shows a Java method `private static HashMap<String, String> getGroupDefinitions(Element e) {`. A red breakpoint is set on the line `for (int i = 0; i < children.size(); i++) {`. The code below it processes XML children and returns a `HashMap`.
- Debugger Window:**
  - Frames:** Shows the current stack frame: `'main'@1 in group 'main'...`.
  - Variables:** Displays the state of local variables:
    - `e = [nu.xom.Element@929]"[nu.xom.Element: pharmacophore.Container]"`
    - `localName = [java.lang.String@1164]"pharmacophore.Container"`
    - `prefix = [java.lang.String@1165]"`
    - `URI = [java.lang.String@1165]"`
    - `attributes = [nu.xom.Attribute[1]@1166]`
    - `numAttributes = 1`
    - `namespaces = null`
    - `children = [nu.xom.Node[16]@1167]`
    - `childCount = 15`
- Console:** Shows the execution progress: `Running: 0 of 1 testReadPcoreDef (org.opencscience.cdk.test.pharmacophore.P...` and `testReadPcoreDef (org.opencscience.cdk.t...`. At the bottom, it shows the system command used to run the application: `/System/Library/Frameworks/JavaVM.framework/Versions/Current/JDK/Home/bin/java -Xdebug -Xrunjdwp:transport=dt_socket,address=127.0.0.1,connected to the target VM, address: 127.0.0.1:63111, transport: socket`

# Unit Testing

- A unit test is a piece of code that tests a specific function
- Each function (or method) in your program should have a unit test
- May have more than one unit test to check different edge cases

*Implement a function to translate 1 letter  
AA codes to 3 letter codes*

- What would you test?

# Unit Testing

- One style of software development suggests writing the test *before* the actual code
  - ▶ Forces you to define what goes in and what goes out
  - ▶ And check that what does come out is what you expected
- Testing frameworks are available for different languages
  - ▶ [JUnit](#) for Java
  - ▶ [CuTest](#) for C
  - ▶ [CppTest](#) for C++
  - ▶ [NUnit](#) for C#
  - ▶ [unittest](#) for Python (comes with the standard library)

# Unit Testing

- Can be a tedious task to add unit tests
- Partly good habit, but can be enforced
  - ▶ Write a test that checks whether each class and method has been tested
  - ▶ Reports on untested methods
- An example is the CDK project
  - ▶ Classes and methods must be *annotated* with the name of the test class and the test method
  - ▶ A special test examines each class for missing or wrong annotations
  - ▶ The report identifies these cases
  - ▶ Allows us to systematically fix things

# General Debugging Principles

## Writing the code

- Be aware of bugs while writing code - place integrity checks at various points
- Always be suspicious of user input - do proper validation
- Use a log file or logging framework
- Use unit tests

## Fixing the code

- Make one change at a time
- Rollback changes if they have no effect
- When a bug is fixed, add a unit test to check for the absence of the bug

# General Debugging Principles

## Writing the code

- Be aware of bugs while writing code - place integrity checks at various points
- Always be suspicious of user input - do proper validation
- Use a log file or logging framework
- Use unit tests

## Fixing the code

- Make one change at a time
- Rollback changes if they have no effect
- When a bug is fixed, add a unit test to check for the absence of the bug

# Outline

- 1 Debugging
- 2 Profiling
- 3 Static Analysis

# What is Profiling?

- Measure detailed performance of you program
  - ▶ Memory usage
  - ▶ CPU consumption (and hence time)
- Modern profilers a provide a large amount of information
- Also generates useful visualization
- Your program can be running remotely and be profiled

**CPU Statistics**

Hot Spots: Alt+1

**Invocation tree**

All threads together: Alt+2  
By threads: Alt+3

Name	Time (ms)	%
com.yourkit.actions.OpenSnapshotAction\$1.run()	24 218	93%
com.yourkit.core.SnapshotManager.addSnapshot(File, Deobfus	24 218	93%
com.yourkit.util.IntLongCompactMap.forEachEntry(TIntLor	12 531	48%
com.yourkit.core.memory.impl.Loader.<init>(Deobfuscator, M	5 609	22%
sun.awt.shell.ShellFolder.exists()	15	0%
com.yourkit.core.SnapshotManager.addSnapshot(File, Deobfuscat	24 218	93%
com.yourkit.ui.ProgressIndicator\$MyThread.run()	24 218	93%

# The Need for Profiling

## Disadvantages

- It's certainly extra work
- Can be slow due to the profilers overhead

## Advantages

- Allows you to identify what part of the program needs improvement
- Saves you from guessing

*... premature optimization is the root of all evil*

C. A. R. Hoare

# The Need for Profiling

## Disadvantages

- It's certainly extra work
- Can be slow due to the profilers overhead

## Advantages

- Allows you to identify what part of the program needs improvement
- Saves you from guessing

*... premature optimization is the root of all evil*

C. A. R. Hoare

# The Need for Profiling

- Though profiling is useful, it should not be your first step
- Your program design should employ appropriate data structures and algorithms
  - ▶ Binary sort instead of heap sort
  - ▶ Hash table instead of an array
  - ▶ Table lookup rather than calculations
- Also consider using the compiler optimization flags
- For some problems manual optimization of code is worthwhile
  - ▶ Memoization
  - ▶ Loop unrolling
  - ▶ Usually important for high-performance computing
- Once all this is exhausted, start up the profiler

# Tools

- [gprof](#) for profiling C/C++ code on Unix systems
- On OS X, [Shark](#) is very useful for C, C++, Java
- [HPROF](#) comes with the JDK
- Eclipse can be used for profiling via [TPTP](#) - difficult to set up
- [valgrind](#) is both a profiler and extremely versatile
- If you have money, [YourKit](#) is fantastic for Java and C#

# Benchmarking vs Profiling

- Very similar ideas
- Profiling is *running code to identify and characterize performance issues in different parts of the code*
- Benchmarking is generally a little higher level - how does the code perform as a whole on a given task or set of inputs
- In general profiling is much more detailed
  - ▶ Identify how long a function takes
  - ▶ Should also tell you how long different parts of the function are contributing to the total execution time of the function

# Simple Benchmarking

- The simplest approach does not require any external tools
- Simply make calls to the languages' timing functions
  - ▶ Python `time.time()`
  - ▶ Java `System.currentTimeMillis()`
  - ▶ C `clock()` - but is low resolution
- General strategy is

```
start_time = ...
do_task
end_time = ...
output (end_time - start_time)
```

# Benchmarking Code

- Interspersing code with timing calls is a hassle
- When you release the code, make sure to comment them out
- Python provides a handy way to time code, using the `timeit` module
  - ▶ Only suitable for small code snippets
- For Java, consider using `Japex` which lets you setup microbenchmarks without putting explicit timing statements into your code
  - ▶ See [here](#) for an example of a project using Japex to measure performance of various cheminformatics tasks
  - ▶ The results are high level - how fast does SMARTS parsing take
  - ▶ Won't tell you whether tokenization is a bottleneck

# Profiling

Name	Time (ms)	
<All threads>	224,581	100%
net.guha.performance.Fingerprinting.main(String[])	224,486	100%
org.openscience.cdk.fingerprint.Fingerprinter.getFingerprint(IAtomContainer)	223,158	99%
org.openscience.cdk.fingerprint.Fingerprinter.getFingerprint(IAtomContainer, AllRingsFinder)	223,158	99%
org.openscience.cdk.fingerprint.Fingerprinter.findPathes(IAtomContainer, int)	219,479	98%
org.openscience.cdk.aromaticity.CDKHueckelAromaticityDetector.detectAromaticity(IAtomContainer)	1,764	1%
org.openscience.cdk.tools.manipulator.AtomContainerManipulator.percieveAtomTypesAndConfigureAtoms(IAtomContainer)	1,392	1%
java.lang.StringBuilder.toString()	123	0%
java.lang.StringBuilder.append(String)	122	0%
java.util.Random.<init>(long)	122	0%
java.lang.StringBuilder.append(int)	48	0%
java.util.Random.nextInt(int)	17	0%
java.lang.ClassLoader.loadClassInternal(String)	17	0%
java.util.HashMap\$Values.iterator()	16	0%
org.openscience.cdk.io.iterator.IteratingMDLReader.hasNext()	1,135	1%
org.openscience.cdk.io.iterator.IteratingMDLReader.<init>(Reader, IChemObjectBuilder)	147	0%

# Benchmarking Caveats

- What are you actually measuring?
- Environment plays an important role
  - ▶ Presence of other processes on the machine
  - ▶ Language (mainly for VM based languages)
- Three broad stages of a micro-benchmark
  - ▶ Prep
  - ▶ Warmup
  - ▶ Run
- Excellent [presentation](#) on the dangers of micro-benchmarks in Java

# Outline

- 1 Debugging
- 2 Profiling
- 3 Static Analysis

# What is Static Analysis?

- Analysis of source code to check for possible run time bugs
- Can also be extended to documentation (Javadocs)
- Handy because it warns you of possible errors that might not be caught at compile time
- Can also lead to cleaner code, by suggesting refactorings
- It's not perfect and can't read your mind
- See IDEA's [inspections](#) to get an idea of what static analysis can catch

# Static Analysis Examples

- If the try succeeds the print statement works fine
- What happens if `getSomeContainer()` throws an exception?
- `container` is undefined, so we dont know!
- Static analysis will catch this error and suggest that you initialize the variable

```
IAtomContainer container;  
try {  
    container = SomeClass.getSomeContainer();  
} except (Exception e) {  
    e.printStackTrace();  
}  
System.out.println("natom = "+container.getAtomCount());
```

# Static Analysis Examples

- If the try succeeds the print statement works fine
- What happens if `getSomeContainer()` throws an exception?
- `container` is undefined, so we don't know!
- Static analysis will catch this error and suggest that you initialize the variable

```
IAtomContainer container;  
try {  
    container = SomeClass.getSomeContainer();  
} except (Exception e) {  
    e.printStackTrace();  
}  
System.out.println("natom = "+container.getAtomCount());
```

# Static Analysis Examples

- The loop is infinite
- Maybe you forgot to set `foundCycle = false` somewhere?
- The behavior of the program might be undefined
- Static analysis can sometimes identify an infinite loop

```
boolean foundCycle = true;
while (foundCycle) {
    List path = GraphTools.getPath();
    if (path.size() == 0) {
        n += 1;
    }
}
```

# Static Analysis Examples

- The loop is infinite
- Maybe you forgot to set `foundCycle = false` somewhere?
- The behavior of the program might be undefined
- Static analysis can sometimes identify an infinite loop

```
boolean foundCycle = true;
while (foundCycle) {
    List path = GraphTools.getPath();
    if (path.size() == 0) {
        n += 1;
    }
}
```

# Static Analysis Examples

- The loop is infinite, since `n = 10` is always true
- Maybe you meant to write `n == 10`?
- Static analysis can catch this

```
int n = 10;
while (n = 10) {
    List path = GraphTools.getPath();
    if (path.size() == 0) {
        n = 0;
        SomeFunction(path);
    }
}
```

# Static Analysis Examples

- The loop is infinite, since `n = 10` is always true
- Maybe you meant to write `n == 10`?
- Static analysis can catch this

```
int n = 10;
while (n = 10) {
    List path = GraphTools.getPath();
    if (path.size() == 0) {
        n = 0;
        SomeFunction(path);
    }
}
```

# Tools

- Many tools for Java and C#
- **PMD** is a tool to analyze Java code and documentation. Example output for the CDK is [here](#)
- **Emma** performs code coverage analysis for Java programs
- For C code there are a number of tool, many commercial. One of the well known free tool is lint and its more modern version [splint](#)