Debugging

Olivier Parcollet

Center for Computational Quantum Physics (CCQ)

Flatiron Institute, Simons Foundation

New York
Docker image for demo

docker pull wentzell/debugging_demo
Outline

• What kind of bugs?
• A selection of (open source) tools
• **Examples.** Tiny pieces of codes, show the tools in action
• Bug prevention.

• Here simple illustrations with (basic) C++
• Many tools usable in Fortran, similar tools in most languages.
• A few C++ specific issues.
Tests!

- How to detect bugs: have a good test suite.
- Test driven development
- It much better to kill bugs in a test than in a production code.

- Test coverage:
  - Are all functions tested? For which parameters?
  - If you do not RUN your function, it is harder to find bugs (but see static analysis, cf later).
  - There are tools to measure that (kcov, …). Not the topic here.
Different kind of bugs

- Out of bound read/write
  - e.g. array of size N, write/read in N+10.
- Memory leak
- Uninitialised variables
- Undefined behaviour
- Logic error
- Infinite loop
- ...

Tools

- **Static analysers**
  - Static analyzers: detect errors in the code without running it.
  - Helps you to write better, cleaner code. Enforce subset of C++.
  - **Compiler itself !**, clang-tidy, cppcheck, …

- **Dynamical analysers**
  - Automated checks of the code while running (check bounds, etc)
    - **Valgrind**: a “virtual” machine, checking each memory access, that each variable is initialized. No recompilation. *Fortran/C++.*
    - **Clang/gcc sanitizers**: compilation options that “instrument” your code automatically to do these checks. (clang/gcc only).

- **Debugger**
  - Execute program step by step: **GDB**, LLDB, …
Static/Dynamical checks

- **Use them preventively**, on your test suite/code
- Systematic use in “Continuous Integration” (Travis, Jenkins)
- **Static analysis**
  - Check all code.
  - Does not require tests
  - False positives possible
- **Dynamic analysis**
  - Require very **good test coverage**
  - Require to run the tests
  - Almost no false positive
Valgrind

• http://valgrind.org

• Best on Linux (does not work well on OS X).

• Various checkers: here we use “memory” (the default one).

• Contains other tools to analyse cache misses, profiling CallGrind, CacheGrind
Clang sanitizers

- Dynamic analysis tools included with clang and gcc

- **Address Sanitizer**  
  `-fsanitize=address`
  detects buffer overflows, memory leaks, use-after-free,…

- **Memory Sanitizer (clang only)**  
  `-fsanitize=memory`
  detects reads of uninitialized memory

- **Undefined Behavior Sanitizer**  
  `-fsanitize=undefined`
  detects undefined behaviors

- **Thread Sanitizer**  
  `-fsanitize=thread`
  detects data races

- Used to systematically detect bugs by Google, Mozilla, …

- Much less memory/runtime overhead than Valgrind!
Availability

- **Linux**
  - Clang (>3.x), gcc (>4.9)
  - All sanitizers

- **Os X**
  - Not all sanitisers
  - Valgrind does not work well.

- **Others : ?**

- **Recommendation : use Linux for debugging**
Examples
**1-Out of bounds !**

- Typical error: read/write out of the memory reserved for an object.

```cpp
#include <vector>

int fun(int n) {
    std::vector<int> vec(100, 1); // a 1d "array" of 100 int init to 1
    vec[n + 50] = 500; // Invalid Write
    return vec[n + 50]; // Invalid Read
}
```

- Similar: use a “dangling” or null pointer, …

- How does it manifest itself?

**Demo**
1-Out of bounds!

- Segfault: the program crashes …
- … or not!
- Hence it can be undetected for a while (or never?).
- We need to find the first error

![Diagram](image)

Segfault: crash
Even worse. No crash, but corrupt memory
OK
1-Out of bounds!

- Debugger is NOT the best tool in this case.
- The code may not crash
- We want a systematic check
- We want the first error

- Valgrind

  → valgrind ./a.out 3

- Sanitizer Address (ASAN)

  → clang++ -fsanitize=address -fno-omit-frame-pointer -g 1-OutOfBounds.cpp
I - Out of bounds

- **Summary:**
  - **Valgrind**
    - Old tool (>15 years). Works with C++, Fortran, etc…
    - Find out of bounds for heap, but not for stack!
    - Can be slow (10x, 50x slower than original code).
  - **Sanitizer Address (ASAN)**
    - More recent tool. C++ specific.
    - Find more cases (stack and heap)
    - Much faster (2x, 3x slower than original code).
2-Invalid-iterator

- A Typical error in C++. Pointer or iterator invalidation.

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v(10, 3); // 10 numbers init to 3
    auto p = v.begin(); // an iterator on the beginning of the vector
    std::cout << *p << std::endl; // print the first value : should be 3
    v.push_back(7); // Add one more number to the vector,
    std::cout << *p << std::endl; // print again the first value ???
}
```
2-Invalid-iterator

• A Typical error in C++. Pointer or iterator invalidation.

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v(10, 3); // 10 numbers init to 3
    auto p = v.begin(); // an iterator on the beginning of the vector
    std::cout << *p << std::endl; // print the first value : should be 3
    v.push_back(7); // Add one more number to the vector,
    std::cout << *p << std::endl; // print again the first value ???
}
```

• Next generation of compiler will detect this.
  **Lifetime** Proposal for std C++.

• Experimental branch of clang.
  ```
  → clang++ -Wlifetime 7-IteratorInvalid.cpp
  ```

https://godbolt.org/z/Z878Dp
3- Memory leak

- Allocate some memory, and ... forget to release it

```cpp
int main() {
    int *g = new int[100];  // allocate the array of 100 int on the heap
    g = nullptr;            // Lost the pointer.

    // ... whatever ...
}
```

- **Effect**: Often none, except if you call such a function a lot, you will run out of memory!

- **Modern C++**: no new/delete. Should not pass code review!

- **Tools**: Valgrind, ASAN (address sanitizer).

  ➔ `valgrind --leak-check=full ./a.out`  
  ➔ `clang++ -fsanitize=address -g Leak.cpp`
4-Uninitialized variables

- What happens if we forgot to initialise something?

```cpp
#include <iostream>

int main(int argc, char** argv) {
    int num = atoi(argv[1]);  // get the first arg and make it from string -> int
    int factorial;  // OOPS!
    for (int i = 1; i <= num; ++i) {
        factorial *= i;
    }
    std::cout << num << "! = " << factorial << "\n";
    return factorial;
}
```

- Static analyser
  - **Compiler** (clang -Wall detects it, but not gcc)
  - **Cppcheck**

- Dynamical analyzer: **Valgrind, memory sanitiser (MSAN)**

- **Limitation**: libraries must be compiled with `-fsanitize=memory`

  `clang++ -g -fsanitize=memory -fno-omit-frame-pointer -fsanitize-memory-track-origins factorial.cpp`
5- Undefined behaviour

• Undefined behavior sanitizer finds many things:
  • Overflow
  • Division by zero
  • Wrong cast e.g. in calling C lib
  • … Undefined Behavior situation in C++…
5- Undefined behaviour

• Integer overflow.

• << is binary shift, we just shifted too much!

```cpp
#include <iostream>
int main(int argc, char** argv) {
    int num = atoi(argv[1]);  // get the first arg and make it from string -> int

    int t = num << 16;
    int r = t * t;
    std::cout << r << std::endl;

    double x = 1/0.0;
    std::cout << x << std::endl;
}
```

→ clang++ -Wall int_overflow.cpp
→ ./a.out 5
0
→ clang++ -Wall -fsanitize=undefined int_overflow.cpp
→ ./a.out 5
int_overflow.cpp:7:18: runtime error: signed integer overflow: 327680 * 327680 cannot be represented in type 'int'
0
6-Logic error

- The logic of the code is flawed, but no automatic tool will find this!

```cpp
#include <iostream>

int main(int argc, char** argv) {
    int num = atoi(argv[1]);  // get the first arg and make it from string -> int

    int factorial = 0;  // Oops!
    for (int i = 1; i <= num; ++i) {
        factorial *= i;
    }
    std::cout << num << "! = " << factorial << "\n";
    return factorial;
}
```

- Let's follow step by step and use the debugger.
- **GDB**: for many compiled languages:
  - C, C++, FORTRAN, Java, Pascal, Ada, D, Go
- **Gdbgui**: a light interface to GDB (usable remotely). Or use an IDE
7- Infinite Loop

- Due to some logic flaw, your code is stuck in an infinite loop!

```c
int main() {
    int c = 0;
    while (1) {
        c += 1;
        // whatever
    }
}
```

- You can run, and attach gdb to your process on the fly

→ gdb ./a.out --pid=PID_OF_THE_PROCESS

Demo
# gdb <program> [core dump]
Start GDB (with optional core dump).

# gdb --args <program> <args...>
Start GDB and pass arguments

# gdb --pid <pid>
Start GDB and attach to process.

set args <args...>
Set arguments to pass to program to be debugged.

run
Run the program to be debugged.

kill
Kill the running program.

## Breakpoints

break <where>
Set a new breakpoint.

delete <breakpoint#>
Remove a breakpoint.

clear
Delete all breakpoints.

enable <breakpoint#>
Enable a disabled breakpoint.

disable <breakpoint#>
Disable a breakpoint.

## Watchpoints

watch <where>
Set a new watchpoint.

delete/enable/disable <watchpoint#>
Like breakpoints.

## GDB cheatsheet - page 1

### Running

<table>
<thead>
<tr>
<th>Command</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>next</code></td>
<td>Go to next instruction (source line) but don't dive into functions.</td>
</tr>
<tr>
<td><code>finish</code></td>
<td>Continue until the current function returns.</td>
</tr>
<tr>
<td><code>continue</code></td>
<td>Continue normal execution.</td>
</tr>
</tbody>
</table>

### Variables and memory

<table>
<thead>
<tr>
<th>Command</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>print/format &lt;what&gt;</code></td>
<td>Print content of variable/memory location/register.</td>
</tr>
<tr>
<td><code>display/format &lt;what&gt;</code></td>
<td>Like „print“, but print the information after each stepping instruction.</td>
</tr>
<tr>
<td><code>undisplay &lt;display#&gt;</code></td>
<td>Remove the „display“ with the given number.</td>
</tr>
<tr>
<td><code>enable display &lt;display#&gt;</code></td>
<td>Enable or disable the „display“ with the given number.</td>
</tr>
</tbody>
</table>

### Examining the stack

<table>
<thead>
<tr>
<th>Command</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>backtrace</code></td>
<td>Show call stack.</td>
</tr>
<tr>
<td><code>where full</code></td>
<td>Show call stack, also print the local variables in each frame.</td>
</tr>
</tbody>
</table>

### Stepping

<table>
<thead>
<tr>
<th>Command</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>step</code></td>
<td>Go to next instruction (source line), diving into function.</td>
</tr>
</tbody>
</table>
# GDB Cheatsheet - Page 2

## Format
- **a**: Pointer.
- **c**: Read as integer, print as character.
- **d**: Integer, signed decimal.
- **f**: Floating point number.
- **o**: Integer, print as octal.
- **s**: Try to treat as C string.
- **t**: Integer, print as binary (\(t = \text{"tw"}\)).
- **u**: Integer, unsigned decimal.
- **x**: Integer, print as hexadecimal.

## Manipulating the Program
- **set var <variable_name> = <value>**: Change the content of a variable to the given value.
- **return <expression>**: Force the current function to return immediately, passing the given value.

## Sources
- **directory <directory>**: Add directory to the list of directories that is searched for sources.
- **list**: Shows the current or given source context. The `filename` may be omitted. If `last` is omitted the context starting at `start` is printed instead of centered around it.
- **list <filename>:<function>**: Shows the current or given source context. The `filename` may be omitted. If `last` is omitted the context starting at `start` is printed instead of centered around it.
- **list <filename>:<line_number>**
- **list <first>,<last>**: Shows the current or given source context. The `filename` may be omitted. If `last` is omitted the context starting at `start` is printed instead of centered around it.

## Informations
- **disassemble**: Disassemble the current function or given location.
- **disassemble <where>**: Disassemble the current function or given location.
- **info args**: Print the arguments to the function of the current stack frame.
- **info breakpoints**: Print informations about the break- and watchpoints.
- **info display**: Print informations about the „displays“.
- **info locals**: Print the local variables in the currently selected stack frame.
- **info sharedlibrary**: List loaded shared libraries.
- **info signals**: List all signals and how they are currently handled.
- **info threads**: List all threads.
- **show directories**: Print all directories in which GDB searches for source files.
- **show listsize**: Print how many are shown in the „list“ command.
- **whatis variable_name**: Print type of named variable.
8-Thread/OpenMP

- Race condition. Can you spot it/them?

```cpp
#include <omp.h>
#include <iostream>
#include <vector>

int main(int argc, char* argv[]) {
    std::vector<double> data(100, 1.0);
    double sum;

    #pragma omp parallel shared(sum, data)
    {
        sum = 0.0;
        #pragma omp for
        for(int i = 0; i < data.size(); i++){
            sum += data[i];
        }
    }
    std::cout << sum << std::endl;
}
```

→ clang++ -fopenmp -g -fsanitize=thread 8-RaceCondition.cpp

- Experimental branch of clang.
Good practices

• Clear, expressive code (Modern C++)
• Code Review!
• Automated Tests (googletest, TDD)
• Version Control (git)
• Static analyzer (clang-tidy)
• Dynamic analyzer (Valgrind, clang sanitizer)
• Logic error. Debugger, step by step. GDB
• Print stuff? Only in the last resort …
How to prevent bugs?
Bug prevention

- Coding style: use a safe subset of C++, avoid dangerous constructions.
- Clear code.
- Code review to enforce this. Human + automated tools (clang-tidy).
- Know your library
- Example:
  - Pointer invalidation of vector<T> (Board).
  - Memory issues with pointers in C/C++. Memory leaks....
Reminder: stack vs heap

- Stack allocation vs heap allocation

```cpp
int f(int a, int b) {
    int i = a;
    double x = a;
    T q = T{a, b}; // // ....
    T* p = new T{a, b}; // // do a lot ....
    delete p;
}
```

- Stack
  - Local variables
  - Destroyed at }

- Heap
  - Outlives }
Issues!

```c
int f(int a, int b) {
    int i = a;
    double x = a;
    T q = T{a,b}; //
    // ....
    T* p = new T{a,b}; //
    // do a lot ....
    delete p;
}
```

Stack

- Issues: who owns p? who is in charge of deleting p?

Heap

- Two potential bugs:
  - Memory leaks: nobody takes care of delete
  - Dangling pointers: use after deallocated. segfault!
Modern C++: no new & delete

- Use **smart pointers**
- **std::unique_ptr** pointer to an object, no other pointer pointing to it
- RAII (Resource Acquisition Is Initialization) pattern.
- Not copy, can be moved.
- No more memory leaks.

```cpp
{  
    T* p = new T{a,b};
    //
    // do a lot ....
    delete p;
}
```

```cpp
{  
    std::unique_ptr<T> p = std::make_unique<T>(a,b);
    //
    // do a lot ....
}
// delete is automatique
```

**Bad**

*Should not pass code review*

**Good**

*Recommended*
Smart pointers

- `std::unique_ptr`
  - One pointer max to each object
  - `p` can not be copied.

- `std::shared_ptr`
  - Multiple pointers
  - With reference counting, like Python
  - When `#ref = 0`, object is destroyed
Conclusion : good practices

- Clear, expressive code (Modern C++)
- Code Review!
- Automated Tests (googletest, TDD)
- Version Control (git)
- Static analyzer (clang-tidy)
- Dynamic analyzer (Valgrind, clang sanitizer)
- Logic error. Debugger, step by step. GDB
- Print stuff? Only in the last resort …
Thank you for your attention!