Debugging and Testing

- Bugs bugs bugs
- Highly optimized code is ugly code
- Remember your priorities:
  - Correctness
  - Flexibility
  - Performance
The “80-20 rule”

“Software developers spend 80% of their time on debugging already written code and 20% on writing new code”
Testing Code

Incremental testing
Make sure each set of lines, routine, set of routines, code etc is working before the complete system is put together

Regression testing
Problem that triggers a bug is included in the test suite
Re-do all tests when changes are made
Unit testing

Test each function and method (unit of code) independently

Vary the input and verify the output state

Framework libraries (JUnit and ports), coverage-checking tools

Debug builds help!

assert, assert, assert
“Extreme Programming”

Construct tools for testing and set up test data before you start writing a new code.

Automate the testing and add new tests as you refine and find bugs.

When you think everything is working, rerun the complete set of tests.

A version of this: “Nightly builds”
Unit tests

- Can be difficult to implement
- Enforce forethought, planning, *structure*
- Key points to test:
  - Erroneous cases – When unit gets wrong input
  - Pathological cases – When unit gets bad or strange input
- Key things to consider:
  - How can a correct output be characterized?
  - How can you catch an incorrect result?
• **Ex.** `assert( size <= LIMIT );`

• `#define NDEBUG to suppress asserts`
  • or compile with `-DNDEBUG`

• **Use to check:**
  • preconditions – Is my function receiving valid parameters?
  • postconditions – Did my function produce reasonable results?
Localizing bugs

Print statements (Not stone age!)

```c
printf("Entering routine xxx-yy\n");
printf("Before main loop in xxx algorithm\n");
printf("%s: %d\n", __FILE__, __LINE__);`
```
Localizing bugs

Print contents of variables and data structures

Write print routines for displaying the contents of your data structures in a readable format.

Print data in condensed form, e.g. the norm of a vector instead of all the vector entries.

Use the more or less tools for examining output, or use a text editor.

Display results graphically.
Localizing bugs

Compare and contrast

A working program in another language, on another system etc is a very valuable resource

BUT: Can the ”working program” be trusted?

Subproblems, or simplified problems, can be easily implemented in e.g. Matlab

Use the same test data for both codes!

Regression testing by giving identical input to “known good” stable version
Localizing bugs

Things to try if you can't find a bug:

- Identify code areas that do NOT contain the bug
- Remove chunks of code until bug disappears
- Fix hacks & kluges, restructure code
- Rewrite the buggy code from scratch
- Write code on paper and execute
Debugging

Find causes of errors in your code
Executes your code like an interpreter
   You can step through your program
   Set breakpoints
   Print and set variables
   Catch run-time exceptions

Two standard debuggers
   GNU gdb
   dbx
Compiling for debugging

To be able to refer to names and symbols in your source code it must be stored in the executable file

Standard flag “-g”

A “debug build” can also enable lower optimization (-O0), debug mode in libraries (asserts)

Without “-g” you can only debug using virtual addresses
Using GDB

```c
int main(void) {
    double pi;
    pi = 4.0 * atan(1.0);
    printf("Pi is %lf\n", pi);
    return 0;
}
```

```
$ gdb ./pi
(gdb) break main
Breakpoint 1 at 0x8048380: file pi.c, line 10.
(gdb) run
Starting program: /home/sverker/test/trunk/pi
Breakpoint 1, main (argc=1 ,..) at debug_me.c:19
(gdb) print pi
$2 = 4.8542713620543657e-270
(gdb) next
11        printf("Pi is %lf\n",pi);
(gdb) print pi
$3 = 3.1415926535897931
(gdb) list
6       int main(void) {
7
8         double pi;
9
10        pi = 4.0*atan(1.0);
11        printf("Pi is %lf\n",pi);
12
13        return 0;
14    }
(gdb) where
#0  main () at pi.c:11
```
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>step [count]</code></td>
<td>Next line of code, step into functions</td>
</tr>
<tr>
<td><code>next [count]</code></td>
<td>Next line of code, execute function</td>
</tr>
<tr>
<td><code>list</code></td>
<td>Print surrounding source</td>
</tr>
<tr>
<td><code>where</code></td>
<td>Print call hierarchy (stack)</td>
</tr>
<tr>
<td><code>print expr</code></td>
<td>Print the value of a variable</td>
</tr>
<tr>
<td><code>break</code></td>
<td>Set breakpoints (many options)</td>
</tr>
</tbody>
</table>
More on GDB

Type `help` in the GDB prompt

`help breakpoints`, `help running`

Check out the gdb man page

GDB can also attach to already running processes owned by you

There is a graphical interface to GDB called the "Dynamic Data Debugger" DDD

$ ddd ./a.out

Allows you to visualize data and breakpoints
Core files

The OS can be configured to dump the virtual address space of a process to a core file

> ulimit -c unlimited

Typically this happens at a fatal error

“Segmentation fault (core dumped)”

This file can be used to debug your program post-mortem

Especially useful is debug symbols are present

> gdb a.out core
Memory bugs

`malloc()` and `free()` – *with friends like these, who needs enemies?*

Can cause problems “far from” where the bug really is
Common to only occur for special data sets
May occur in a seemingly non-deterministic way
Influence from the environment (system load etc)
May be found in codes that were supposed to be “tested and verified”, e.g. when porting to another system
Memory bugs

Unallocated memory

double *A;
....
for (i=0; i<n; i++)
    A[i] = 0.0;

Can cause the execution to stop when the variable is used
or an attempt to deallocate the memory is made

Most compilers can check for unallocated memory
Overwriting or “overreading” memory

Common error in C and Fortran: Using arrays outside the array bounds

- double * A = (double*) malloc(N*sizeof(double));
- ...
- for( i=0; i <= N; i++)
  - Diff[i]=(A[i+1]-A[i])/h;

May result in erroneous results or program crash (page fault in invalid page)

Program crash may occur later, e.g. when calling memory allocation routine

Many compilers can generate code that checks array bounds during execution (add option for this). NOTE: This reduces performance!

Handled in Java and e.g. Pascal (Runtime error)
Memory bugs

Stack buffer overrun/overflow

Special case of memory overreading/overwriting where you run out of stack space.

Caused by:

- Too many very large stack variables
- Too many (recursive) function calls
Memory bugs

Dangling pointers

A pointer to a piece of memory that once was allocated to a variable, but then was deallocated

Common error when copying and deallocating derived types with dynamically allocated fields

Another common error: returning a pointer to a local variable

Hard to debug! Some debugging tools may help you

0xBAADF00D, 0xDEADBEEF
Memory bugs

Memory leaks

Allocated memory is lost without deallocation

The program runs out of memory for large problems, many iterations …

Hard to detect. Use e.g. `top` to monitor memory usage

Hard to debug. Instrumented system call libraries may give some information; allocation “tagging”.

Valgrind memcheck can help

Handled in e.g. Java, and other languages with built-in garbage collection
Memory bugs and testing

- Tests often miss memory bugs or
- Tests only *sporadically* catch memory bugs
- “Heisenbugs” stop failing when run in gdb
  - pointer initialization usually the culprit
- What to do?
  - Check memory addresses
  - Isolate program components
  - Use Valgrind
Valgrind

- Use of uninitialised memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks -- where pointers to malloc'd blocks are lost forever
- Mismatched use of malloc/new/new [] vs free/delete/delete []
- Overlapping src and dst pointers in memcpy() and related functions
- Some misuses of the POSIX pthreads API

% valgrind --tool=memcheck program_name
...
=18515== malloc/free: in use at exit: 0 bytes in 0 blocks.
==18515== malloc/free: 1 allocs, 1 frees, 10 bytes allocated.
==18515== For a detailed leak analysis, rerun with: --leak-check=yes
Valgrind caveats

Valgrind will NOT perform bounds checking on static arrays (allocated on the stack)

X 2 memory usage

Program will have terrible performance

Valgrind may not find all the errors – especially those that don't happen every time.
Computational bugs

Floating-point problems
  Cancellation
  Precision (using numbers of very different magnitude)
  Accuracy of intrinsic functions and constants (embedded code, GPUs)
  Strict equality is always dangerous. Always use:
  \( \text{fabs}(x-y) < \text{EPSILON} \), \textbf{not} \( x == y \)
  Ordering is crucial!

Algorithms used near their limit of applicability
  Near-indefinite matrix
  Near-singular matrix
  Near-degenerate geometry in computational grid

Stress testing: Include “extreme inputs” in test data set
Computational bugs

Overflow/underflow errors

Ex. get middle array element in a segment of a very large array:

```java
int middle = (left + right)/2; //DANGEROUS
```
Logical bugs and typos

Just about anything…

Mixing up negation

Assigning when intending equality

Misusing a library, or another part of your code

"That case will never arise"

"I use this argument in a clever way to indicate two things"
What to do?

Tedious code is tedious to test
Respect compiler warnings (use \texttt{-Wall})

\emph{Reusing} code is good, \emph{copying} code is a slippery slope

Use the libraries of others:
Data structures, common routines & operations
\texttt{e.g. Boost (www.boost.org)}