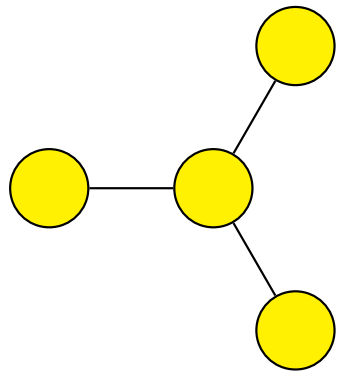
C PROGRAMMING TOOLS

part of the PROGRAMMING III course



Evangelos Ververas e.ververas16@ic.ac.uk
Duncan White d.white@imperial.ac.uk

Pedro Mediano

Table of Contents

DEBUGGING

THE GNU DEBUGGER: **gdb**

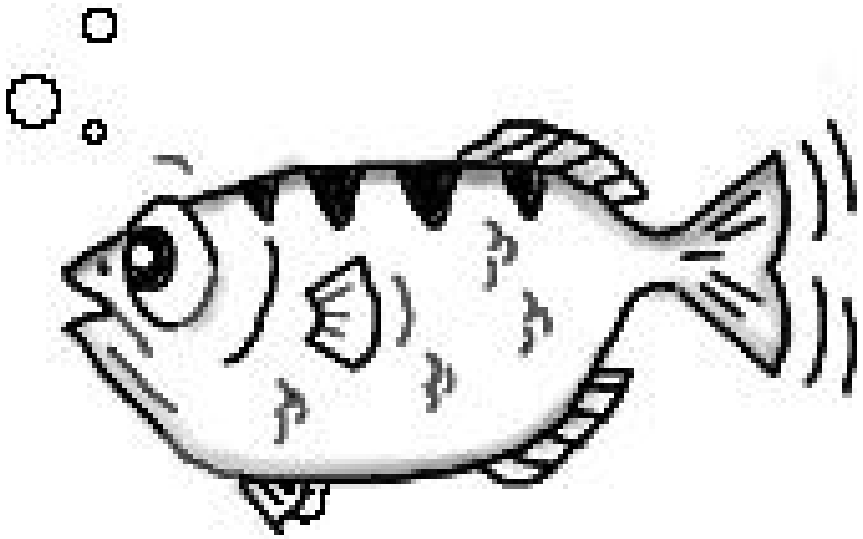
According to GNU, a debugger:
Allows you to see what is going on 'inside' another program while it executes – or what another program was doing at the moment it crashed.

THE GNU DEBUGGER: **gdb**

According to GNU, a debugger:
Allows you to see what is going on 'inside' another program while it executes – or what another program was doing at the moment it crashed.

Typical debugging cycle:

1. Start the program.

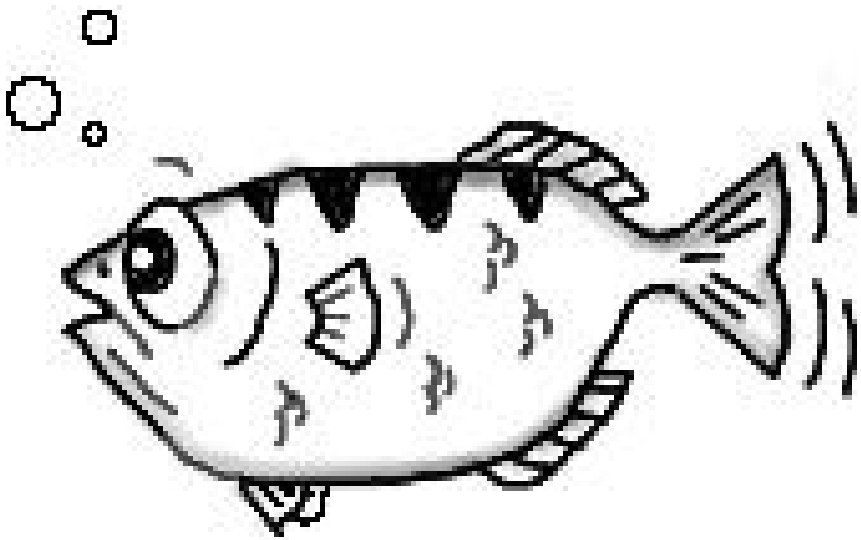


THE GNU DEBUGGER: **gdb**

According to GNU, a debugger:
Allows you to see what is going on 'inside' another program while it executes – or what another program was doing at the moment it crashed.

Typical debugging cycle:

1. Start the program.
2. Stop execution.

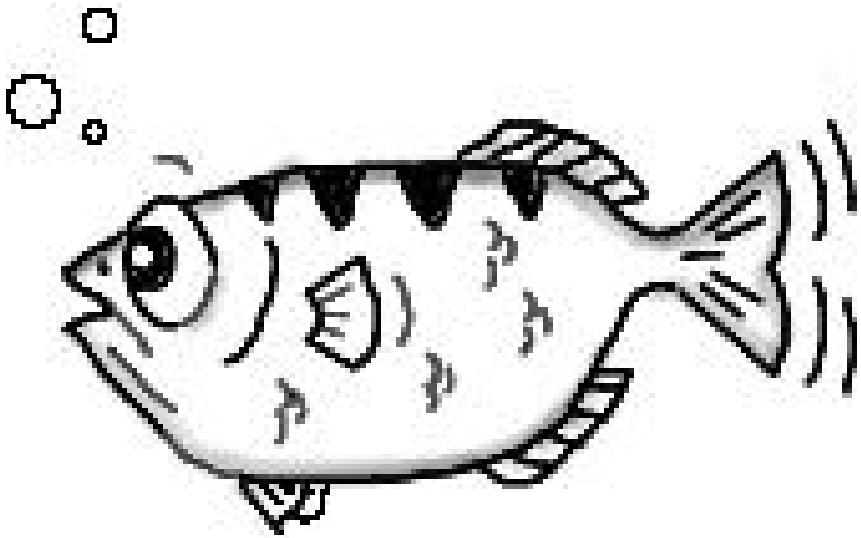


THE GNU DEBUGGER: **gdb**

According to GNU, a debugger:
Allows you to see what is going on 'inside' another program while it executes – or what another program was doing at the moment it crashed.

Typical debugging cycle:

1. Start the program.
2. Stop execution.
3. Diagnose problem.

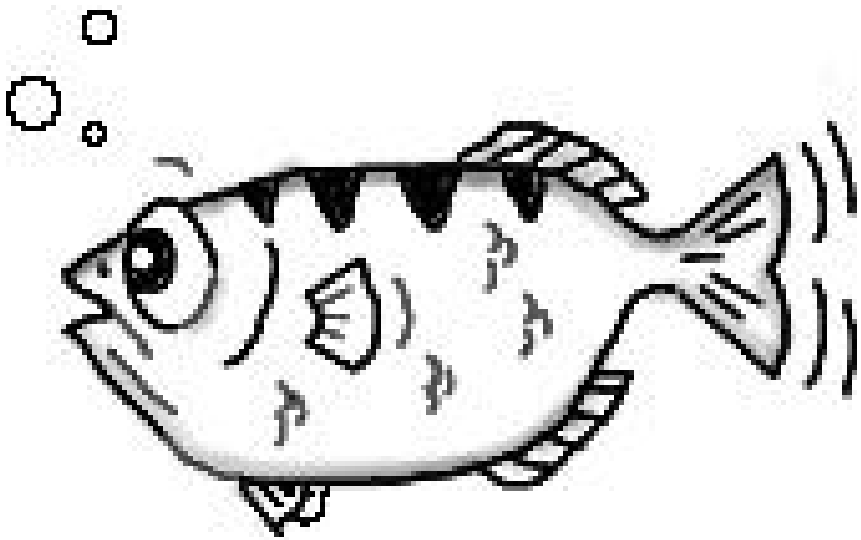


THE GNU DEBUGGER: **gdb**

According to GNU, a debugger:
Allows you to see what is going on 'inside' another program while it executes – or what another program was doing at the moment it crashed.

Typical debugging cycle:

1. Start the program.
2. Stop execution.
3. Diagnose problem.
4. Re-write the code.

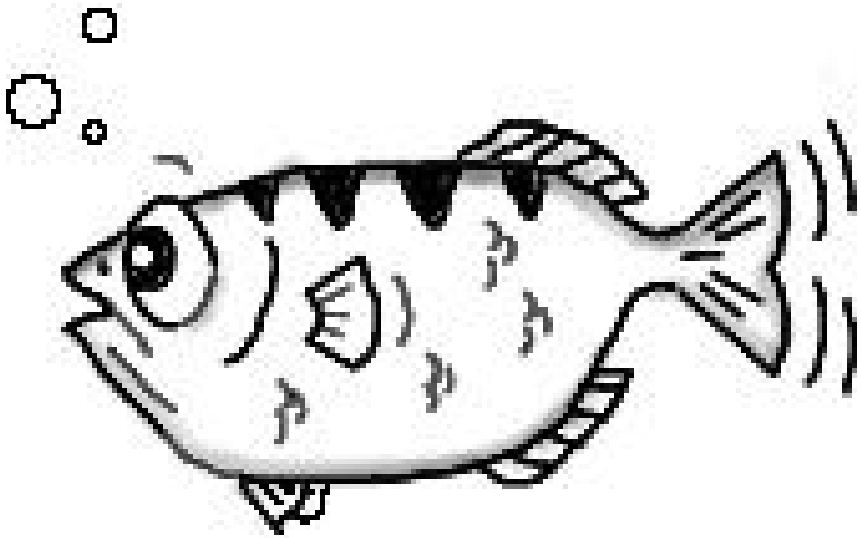


THE GNU DEBUGGER: **gdb**

According to GNU, a debugger:
Allows you to see what is going on 'inside' another program while it executes – or what another program was doing at the moment it crashed.

Typical debugging cycle:

1. Start the program.
2. Stop execution.
3. Diagnose problem.
4. Re-write the code.
5. Go back to step 1.



A QUICK **gdb** CHEATSHEET

`r`	Execute the program loaded
`bt` or `where`	Print the call frame stack
`b [N|FUNC]`	Break at line `N` or at the start of function `FUNC`
`l`	Print code around current location
`p [VAR]`	Print contents of variable `VAR`
`x [ADDR]`	Examine contents of address `ADDR`
`watch [VAR]`	Break whenever variable `VAR` is written
`help`, `quit`	Hopefully self-explanatory

Don't forget to use &

You might like `cgdb` or `gdb -tui`

MEMORY PROBLEMS

MEMORY ISSUES

Memory problems are the most **serious C problems**:

$\rightarrow$ Often claimed that 99% of serious C bugs are memory-allocation related.

MEMORY ISSUES

Memory problems are the most **serious C problems**:

$\rightarrow$ Often claimed that 99% of serious C bugs are memory-allocation related.

WHY IS THAT?

In general, **C does not care** and lets you play with memory at will.

MEMORY ISSUES

Memory problems are the most **serious C problems**:

→ Often claimed that 99% of serious C bugs are memory-allocation related.

WHY IS THAT?

In general, **C does not care** and lets you play with memory at will.

REALLY BAD STUFF

Things you **REALLY** shouldn't do:

- Not checking array bounds.

```
int array[4], i;
for (i=0;i<10;i++)
  array[i] = 0;
```

REALLY BAD STUFF

Things you **REALLY** shouldn't do:

- Not checking array bounds.
- Dereferencing null pointers.

```
int *ptr = NULL;
*ptr = 1;
```

REALLY BAD STUFF

Things you **REALLY** shouldn't do:

- Not checking array bounds.
- Dereferencing null pointers.
- `free()` something twice.

```
int *p =
  malloc(5*sizeof(int));
free(p);
free(p);
```

REALLY BAD STUFF

Things you **REALLY** shouldn't do:

- Not checking array bounds.
- Dereferencing null pointers.
- `free()` something twice.
- Causing a stack overflow.

```
int main(){
  main();
  return 0;
}
```

REALLY BAD STUFF

Things you **REALLY** shouldn't do:

- Not checking array bounds.
- Dereferencing null pointers.
- `free()` something twice.
- Causing a stack overflow.
- Writing read-only memory.

```
char *s = "get ready";
*s = 'x';
```

REALLY BAD STUFF

Things you **REALLY** shouldn't do:

- Not checking array bounds.
- Dereferencing null pointers.
- `free()` something twice.
- Causing a stack overflow.
- Writing read-only memory.

Common result of these is a

Segmentation fault

MEMORY LEAKS

Always `free()` what you `malloc()`!

MEMORY LEAKS

Always **`free()`** what you **`malloc()`**!

- Unfree'd memory will remain useless.
- Leaky programs might eat your whole RAM;
- And they're usually slower.

MEMORY LEAKS

Always **`free()`** what you **`malloc()`**!

- Unfree'd memory will remain useless.
- Leaky programs might eat your whole RAM;
- And they're usually slower.

There are lots of tools out there:
→ **`valgrind`**, **`libmem`**

PROFILING

PROFILING

DEFINITION

Profiling *is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls.*

PROFILING

DEFINITION

Profiling *is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls.*

Make yourself useful:

→ Find the hot spots that really need optimizing.

→ **Never** start optimizing before profiling.

SIMPLE "PROFILING" TOOLS

Two blunt, yet accessible tools for performance analysis:

SIMPLE "PROFILING" TOOLS

Two blunt, yet accessible tools for performance analysis:

`htop`

- ✓ Monitor memory and CPU usage in real-time.
- ✗ All processes are mixed up.

SIMPLE "PROFILING" TOOLS

Two blunt, yet accessible tools for performance analysis:

`htop`

- ✓ Monitor memory and CPU usage in real-time.
- × All processes are mixed up.

`time`

- ✓ Measure user, kernel and system execution time.
- × Not very accurate.

SIMPLE "PROFILING" TOOLS

Two blunt, yet accessible tools for performance analysis:

`htop`

- ✓ Monitor memory and CPU usage in real-time.
- × All processes are mixed up.

`time`

- ✓ Measure user, kernel and system execution time.
- × Not very accurate.

`gcc -pg / gprof`

- ✓ The only proper C profiler for gcc.

UNDERSTANDING THE PROFILER

There are (mostly) three things profilers can do:

FLAT PROFILE
: Show time spent in each function and total number of calls.

CALL GRAPH
: Build a who-calls-who diagram of all functions.

ANNOTATED SOURCE
: Display source with a line-by-line execution count.

UNDERSTANDING THE PROFILER

There are (mostly) three things profilers can do:

FLAT PROFILE
: Show time spent in each function and total number of calls.

CALL GRAPH
: Build a who-calls-who diagram of all functions.

ANNOTATED SOURCE
: Display source with a line-by-line execution count.

Explore some profilers:

→ **`gprof`**, **`callgrind`**, **`perftools`**

And use a profile visualization tool:

→ **`kcachegrind`**

MEASURING TIME

Common profilers (e.g. **`gprof`**) usually measure *user time*, instead of *kernel* or *wall clock* time.

MEASURING TIME

Common profilers (e.g. **`gprof`**) usually measure *user time*, instead of *kernel* or *wall clock* time.

- ✓ Not affected by other irrelevant processes
- × Useless if program spends most of the time in kernel.

MEASURING TIME

Common profilers (e.g. **gprof**) usually measure *user time*, instead of *kernel* or *wall clock* time.

- ✓ Not affected by other irrelevant processes
- ✗ Useless if program spends most of the time in kernel.

```c
#include <stdlib.h>

void siesta(void) {
  sleep(5);
  return;
}

int main() {
  unsigned int i;
  for (i=0; i<100; i++) {
    siesta();
  }
  return 0;
}
```

TESTING

TESTING

- Pragmatic programmer's tip 62:
  Tests that run with every build are much more effective than test plans that sit on a shelf.

TESTING

- Pragmatic programmer's tip 62:
  Tests that run with every build are much more effective than test plans that sit on a shelf.

- Corollary:
  Automate your tests.

TESTING

- Pragmatic programmer's tip 62:
  Tests that run with every build are much more effective than test plans that sit on a shelf.

- Corollary:
  Automate your tests.

We'll talk about automatic tests in
`make`, **`cmake`**, **`git`**

AUTOMATED TESTING

- Git hooks:
  Run scripts before **`push`**/**`commit`**.
  Good moment for style checks (e.g. Google's **`cpplint`**)

AUTOMATED TESTING

- Git hooks:
  Run scripts before **`push`**/**`commit`**.
  Good moment for style checks (e.g. Google's **`cpplint`**)

- Makefile tests:

```
test:  testprogram1 testprogram2 ...
    ./testprogram1
    ./testprogram2
```

AUTOMATED TESTING

- Git hooks:
  Run scripts before **`push`**/**`commit`**.
  Good moment for style checks (e.g. Google's **`cpplint`**)
- Makefile tests:

```
test:  testprogram1 testprogram2 ...
    ./testprogram1
    ./testprogram2
```

- CMake tests:

```
add_test(PreliminaryTest testlist)
```

- Other testing modules (e.g. C++ **`boost`**).

TEST-DRIVEN DEVELOPMENT

Basic principle:

Write the test before the function.

TEST-DRIVEN DEVELOPMENT

Basic principle:

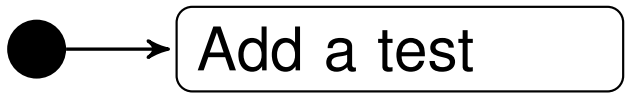
Write the test before the function.

- Helps measuring progress.
- Encourages modularity and extensibility.
- Reduces debugger use.
  - But if you find a new bug, write a test for it!
- Don't forget to add some overall tests.

TEST-DRIVEN DEVELOPMENT

Basic principle:

Write the test before the function.

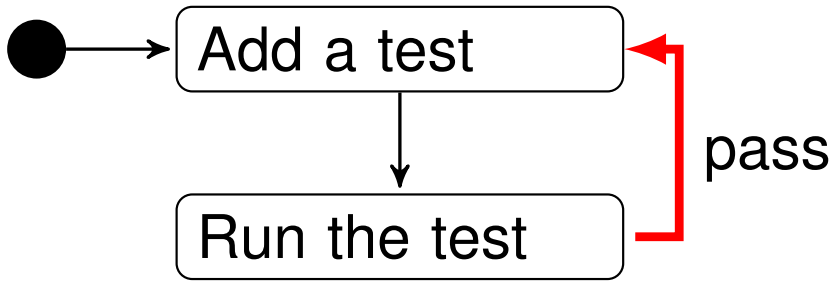


- Helps measuring progress.
- Encourages modularity and extensibility.
- Reduces debugger use.
  - But if you find a new bug, write a test for it!
- Don't forget to add some overall tests.

TEST-DRIVEN DEVELOPMENT

Basic principle:

Write the test before the function.

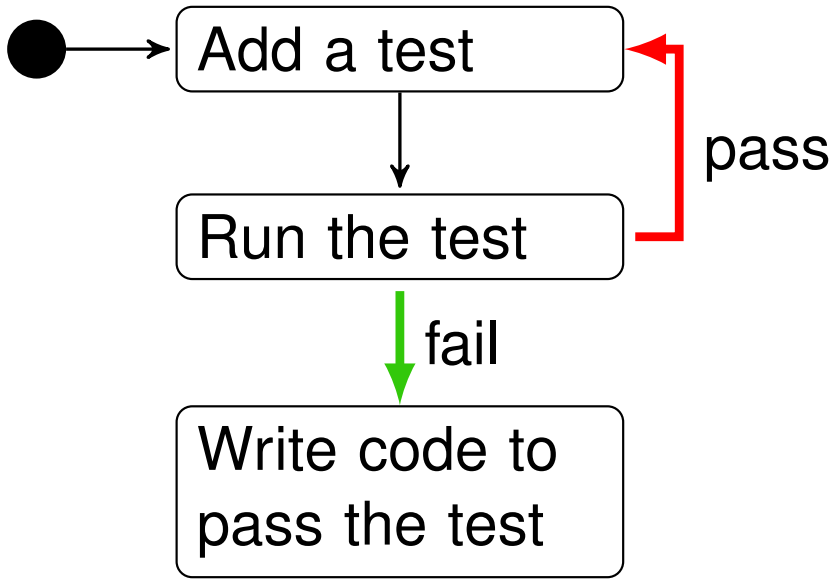


- Helps measuring progress.
- Encourages modularity and extensibility.
- Reduces debugger use.
  - But if you find a new bug, write a test for it!
- Don't forget to add some overall tests.

TEST-DRIVEN DEVELOPMENT

Basic principle:

Write the test before the function.



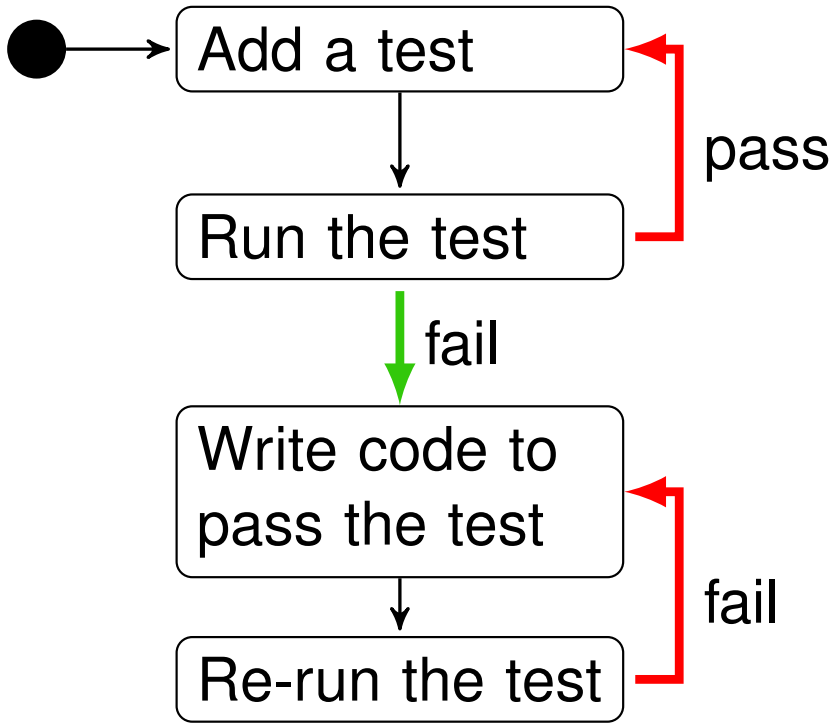
- Helps measuring progress.
- Encourages modularity and extensibility.
- Reduces debugger use.
  - But if you find a new bug, write a test for it!
- Don't forget to add some overall tests.

TEST-DRIVEN DEVELOPMENT

Basic principle:

Write the test before the function.

- Helps measuring progress.
- Encourages modularity and extensibility.
- Reduces debugger use.
  - But if you find a new bug, write a test for it!
- Don't forget to add some overall tests.



Test-driven development

Basic principle:

Write the test before the function.

- Helps measuring progress.
- Encourages modularity and extensibility.
- Reduces debugger use.
  - But if you find a new bug, write a test for it!
- Don't forget to add some overall tests.

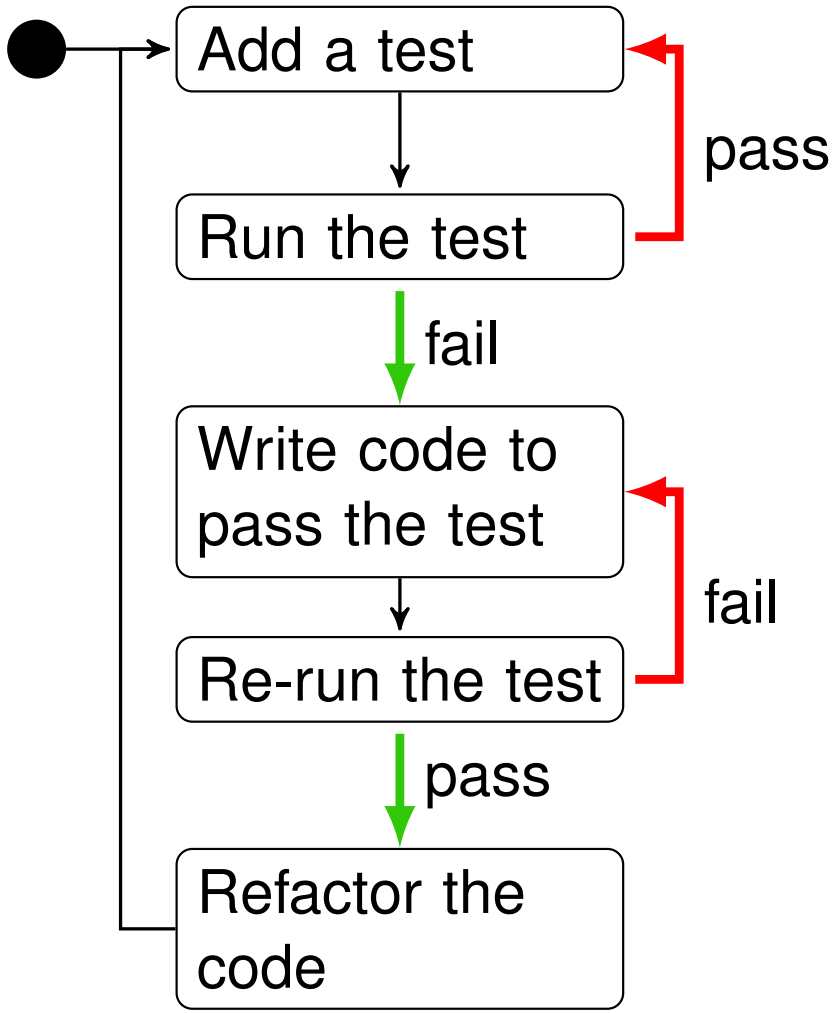


TABLE OF CONTENTS