Chapter 3

Development Environment

3.1 Prerequisites

Before exploring this chapter, ensure you have access to a standard computer—whether a Mac, PC, or Linux machine—along with a text editor, a C compiler, and a Python interpreter. A strong proficiency in both C and Python is assumed, as we will not cover their fundamentals. Rather than focusing on integrated development environments (IDEs), our attention will be directed toward the core components that underpin these tools, including debugging, profiling, and other essential aspects of program analysis.

3.2 Basic tools

Computers have inherent limitations, such as processing power, memory capacity, and input/output capabilities, which can significantly affect program performance, particularly with large datasets or demanding computations. Consequently, programmers must strive for not only bug-free code but also robust and optimised solutions.

Stability is critical; a stable program behaves predictably under various conditions, which is vital in fields like healthcare and finance. Optimisation plays a key role in enhancing efficiency, reducing resource consumption, and improving user experience. This includes writing efficient algorithms, optimising memory usage, and ensuring a responsive user interface.

Successful programming requires a holistic approach that encompasses thorough testing, debugging, and a deep understanding of the software's context. As technology evolves, programmers must remain adaptable and continuously refine their skills to meet the challenges of creating efficient and effective software

solutions.

3.3 Debugging

Let's first explore the three common types of bugs in programming: syntax errors, logical errors, and runtime errors.

Syntax errors

These occur when the code violates the rules or grammar of the programming language. They are caught by the compiler or interpreter before the program runs.

```
#include <stdio.h>

int main() {
    printf("Hello, World!"
    return 0;
}
```

Here, there's a missing closing parenthesis ')' in the 'printf()' statement, and also a semicolon ';' after. When the code is compiled, the compiler will generate an error, indicating that the syntax is incorrect. Syntax errors prevent the program from running at all.

How they manifest:

- The program will not compile or run.
- The compiler/interpreter will provide an error message pointing out the location of the issue.

Logical errors

Logical errors occur when the program runs, but it produces incorrect or unintended results. These errors are caused by flawed logic in the program, and they are the hardest to detect because the code doesn't crash or halt. It simply gives the wrong output.

```
#include <stdio.h>

int main() {
    int a = 10;
    int b = 20;
```

```
    int sum = a * b; // Should be a + b instead of a * b
    printf("Sum is: %d\n", sum);
    return 0;
}
```

Here, the programmer intended to calculate the sum of a and b, but instead, the product is calculated due to using the wrong operator (* instead of +). This is a logical error because the program runs successfully but produces an incorrect result: “Sum is: 200” instead of “Sum is: 30”.

How they manifest:

- The program runs without errors.
- The output or behaviour of the program is incorrect or unexpected.
- The compiler cannot catch this kind of bug since the syntax is valid.

Runtime errors

Runtime errors occur while the program is running. These errors are typically caused by invalid operations like dividing by zero, accessing memory that doesn’t exist, or using resources improperly. These errors cause the program to crash or terminate unexpectedly.

```
#include <stdio.h>

int main() {
    int x = 10;
    int y = 0;
    int result = x / y; // division by zero
    printf("Result: %d\n", result);
    return 0;
}
```

In this case, trying to divide x by y results in a division-by-zero error, which is undefined behaviour (in most programming languages). This runtime error will cause the program to crash.

How they manifest:

- The program starts but crashes or halts unexpectedly.
- The error occurs only during execution, often accompanied by an error message (e.g., “Segmentation fault” or “Division by zero”).

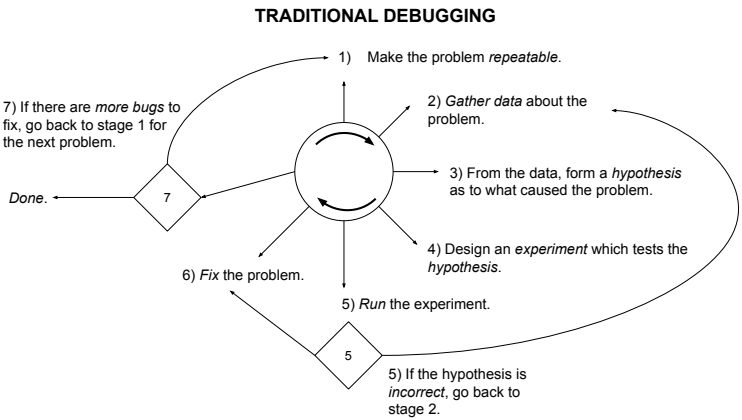
Summary

- Syntax errors are detected by the compiler or interpreter before execution, preventing the program from running.
- Logical errors occur when the program runs but produces incorrect results due to faulty logic.
- Runtime errors occur during program execution, causing the program to crash or behave unexpectedly.

Understanding the differences between these types of bugs helps in identifying and fixing errors more effectively during the development process.

3.3.1 Process

Debugging is the process of identifying and resolving defects or issues in a program. Certain bugs are easily identified and resolved, but some can be persistent and very hard to spot.



The stages of non-trivial debugging are:

1. Ensure the problem can be *consistently reproduced*. This typically involves identifying which specific inputs trigger the issue.
2. Collect *detailed data* about the problem. This step is essential, so approach it with patience and attention to detail.

3. Based on the gathered data, *develop a hypothesis* about the underlying cause of the problem. This educated guess should be informed by the patterns or clues found in the data.

4. Create an experiment that directly *tests the hypothesis*. The key is to structure the experiment so that it yields a clear yes or no result, confirming or disproving the hypothesis.

5. *Run the experiment*. If the hypothesis is incorrect, go back to stage 2.

6. Resolve the problem by *applying the appropriate solution* based on the outcome of the experiment. Ensure that the fix addresses the root cause and thoroughly test it to verify the issue is fully resolved.

7. If there are *additional bugs* to resolve, return to stage 1 and repeat the process for each new problem. This ensures a systematic and thorough approach to debugging.

To effectively tackle a bug or issue, the first step is to make the problem repeatable. This typically requires identifying the specific inputs or conditions that consistently lead to the issue, ensuring that the problem can be triggered reliably. Once this is achieved, it's essential to gather as much data as possible about the problem. Carefully observe any error messages, crash dumps, or logs, as these often contain valuable clues that should not be overlooked. This stage requires patience and attention to detail, as premature conclusions can lead to missed insights.

With the data in hand, the next step is to form a hypothesis about the root cause of the problem. Based on the information gathered, you can then design an experiment to test the hypothesis. The key here is to ensure that the experiment produces a clear, binary result—either the hypothesis is supported or it is disproven. After running the experiment, if the hypothesis turns out to be incorrect, it's crucial to return to the data-gathering phase and refine your understanding before trying again.

Once the correct hypothesis has been confirmed, the issue can be fixed. If multiple bugs remain, the process starts anew, following the same structured approach. This methodical, iterative process not only helps solve the current problem but also improves debugging skills by reinforcing a disciplined, logical approach to problem-solving.

3.3.2 Root Cause Analysis

Root cause analysis (RCA) is a systematic approach to identifying the fundamental cause of problems rather than merely addressing their symptoms. In debugging

contexts, RCA ensures that fixes target the underlying issue, preventing recurrence and building more robust systems. This methodology complements the debugging process outlined in Section 3.3.1 by providing structured techniques for the hypothesis formation and testing stages.

The distinction between symptoms and root causes is basic in effective debugging. A *symptom* is the *observable manifestation of a problem*—such as a program crash, incorrect output, or performance degradation—while the *root cause* is the fundamental *reason (or cause) why the problem occurs.* For instance, if a web application frequently times out, the symptom is the timeout, but the root cause might be inefficient database queries, inadequate connection pooling, or insufficient server resources.

Five Whys Technique

The Five Whys technique involves asking "why" repeatedly to drill down from symptoms to root causes. Starting with the problem statement, each answer becomes the basis for the next "why" question, typically requiring three to five iterations to reach the fundamental cause.

Debugging example.

- Problem: Program crashes with segmentation fault
- Why? Attempting to access invalid memory address
- Why? Array index exceeds bounds
- Why? Loop counter increments beyond array size
- Why? Loop condition uses <= instead of <
- Why? Programmer misunderstood array indexing (off-by-one error)

The root cause isn't the segmentation fault itself, but the conceptual error in array indexing logic.

Best practices for Five Whys:

- Focus on process failures rather than individual blame
- Involve team members with direct knowledge of the system
- Don't stop at the first obvious answer—continue probing
- Document each step for future reference and learning
- Verify the root cause by confirming that addressing it prevents recurrence

Fishbone Diagram (Ishikawa Diagram)

The fishbone diagram provides a visual framework for systematically exploring multiple potential causes. Named for its fish-like appearance, this tool organises potential causes into categories, helping teams brainstorm comprehensively rather than fixating on obvious suspects.

Common categories for software debugging:

- *Code.* Logic errors, syntax issues, algorithmic flaws
- *Data.* Input validation, data corruption, format mismatches
- *Environment.* Configuration, dependencies, hardware limitations
- *Process.* Development workflows, code review gaps, testing inadequacies
- *People.* Skill gaps, communication failures, requirement misunderstandings
- *Tools.* Compiler bugs, library incompatibilities, development environment issues

Implementation process:

1. Draw the problem statement in a box on the right
2. Draw a horizontal line (spine) pointing to the problem
3. Add diagonal lines (bones) for each category
4. Brainstorm specific potential causes under each category
5. Analyse relationships between causes
6. Prioritise the most likely root causes for investigation

See an example for "Memory leak in long-running application" in Figure 3.1.

Integrating RCA with Debugging Workflow

Root cause analysis integrates naturally with the debugging process described in Section 3.3.1:

1. **Problem reproduction.** Establish clear symptoms and reproduction steps
2. **Data collection.** Gather comprehensive information about the failure
3. **RCA application.** Use Five Whys or fishbone diagrams to systematically explore causes

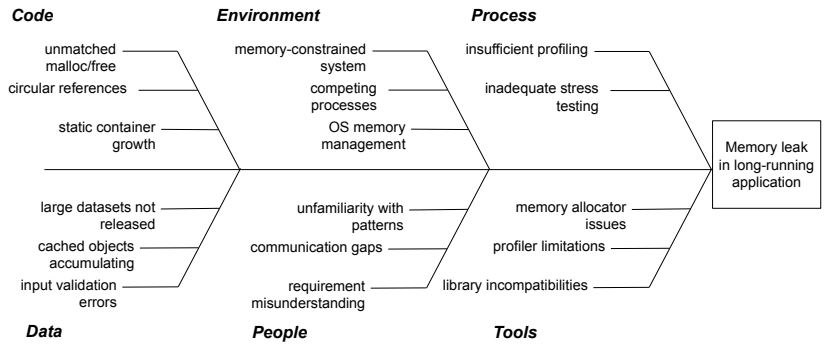


Figure 3.1: Fishbone diagram for debugging memory leak root causes. Each category branch contains specific potential causes that could contribute to the problem.

4. **Hypothesis formation.** Generate testable hypotheses based on RCA findings
5. **Experimentation.** Test hypotheses to confirm or refute potential root causes
6. **Solution implementation.** Address the identified root cause, not just symptoms
7. **Verification.** Confirm that the solution prevents problem recurrence

Both RCA techniques work synergistically—use fishbone diagrams for comprehensive cause exploration, then apply Five Whys to drill deeper into the most promising areas. This combination ensures thorough investigation while maintaining focus on actionable solutions.

The systematic nature of root cause analysis makes it particularly valuable for complex bugs, recurring issues, and post-incident reviews. By documenting RCA findings, teams build institutional knowledge that helps prevent similar problems and improves overall system reliability.

3.3.3 Tools

We present a selection of tools that, while simple and perhaps somewhat basic, serve as useful starting points. Although there are many tools worth exploring for debugging and related tasks, we will focus on just a few key examples.

The following highlights open-source tools commonly used in Linux, such as GDB and GCC, which are also compatible with Microsoft Windows and macOS. Additionally, commercial equivalents of these tools are available for both Windows and macOS platforms.

Print statements

One of the simplest and earliest debugging techniques is inserting print statements into code to monitor its execution at runtime. This method provides visibility into variable values, program flow, and state at various points in the code.

Developers insert lines like 'print()' in Python or 'printf()' in C at critical places in the code to output values to a console or log file. For example, you might print the state of a loop counter, input values, or function return values.

```
int main() {
    for (int i = 0; i < 10; i++) {
        printf("Iteration %d\n", i);
    }
    return 0;
}
```

- Pros:
  - Extremely simple to implement.
  - No special tools or setup is required.
  - Great for quick checks on values and flow.
- Cons:
  - Invasive as it requires changing the code itself, and too many print statements can clutter the output.
  - Doesn't work well for timing bugs or complex interactions, especially in real-time systems.

Logging

Logging is a more structured version of print debugging where developers log critical events, data, and state to a file for post-execution analysis. Historically, this was done with simple file output, but logging libraries have since become common to handle this in a more organised way.

Instead of using print statements, you can use logging frameworks to log messages, variable values, and error states to a file or output stream, often with different severity levels (e.g., INFO, DEBUG, ERROR).

```
import logging

# set up logging configuration
logging.basicConfig(level=logging.DEBUG,
                format='%(asctime)s - %(levelname)s - %(message)s')

def example_function():
    logging.debug('This is a debug message.')
    logging.info('This is an info message.')
    logging.warning('This is a warning message.')
    logging.error('This is an error message.')
    logging.critical('This is a critical message.')

if __name__ == '__main__':
    example_function()
```

- Pros:
  - Non-intrusive as logs can be left in the code and turned on/off as needed.
  - Works well in production environments where real-time debugging isn't possible.
  - Logs provide a timeline of program execution, helping diagnose complex, asynchronous, or time-sensitive issues.
- Cons:
  - Can introduce performance overhead if logging is too verbose or frequent.
  - Requires careful planning of what to log to avoid overwhelming the developer with data.

LEDs and serial ports

In the early days of embedded systems (and still common today), developers would use serial output or LEDs to debug their hardware programs. When running on hardware, especially microcontrollers with limited output capability, toggling an LED or sending data over a serial port provided a way to communicate program state.

Example was to blink an LED if a certain part of the program is reached, or use a serial port to send debug information (e.g., UART communication, see Section **??**).

```
gpio_set_function(25, GPIO_OUT); // Raspberry Pi Pico example
gpio_put(25, 1); // turn LED on when certain conditions are met
```

- Pros:
  - Non-intrusive to program logic; you can toggle LEDs or send serial data without altering the main code structure.
  - Works well on hardware with no display or debugging interfaces.
- Cons:
  - Limited in terms of the amount of information conveyed (e.g., toggling an LED is binary).
  - Less useful in many cases for debugging complex software states or logic errors.

Breakpoints and single stepping

Breakpoints and single stepping are classic debugging techniques used with early debuggers, such as the 'GNU Debugger' (GDB), 'dbx' on Unix, or hardware debuggers (see 'Hardware debugging,' p.80). Breakpoints allow you to pause program execution at specific points, while single stepping lets you execute one instruction at a time.

1. A breakpoint is set at a specific line or address in the program, causing the program to halt execution when that point is reached.
2. Once the program is halted, you can inspect the state of variables, memory, and CPU registers.
3. Single stepping then allows you to execute one instruction at a time, monitoring how the state of the system changes.

Breakpoint example. After starting GDB with your compiled program, you can set a breakpoint at the main function. This halts the program's execution at the very beginning, allowing you to inspect the environment before any code has been executed.

```
..
(gdb) break main
(gdb) run
```

Once the breakpoint is hit, you have a wide array of commands available to examine the program's state. For example:

- `info registers`: Shows the current values in the CPU's registers, which is useful for low-level debugging.
- `info locals`: Displays the current values of local variables in the scope of the function where the program is halted.
- `print var`: Allows you to print the value of any variable in scope by using its name (replace var with the actual variable name).
- `step` and `next`: Step into a function or advance to the next line of code, respectively.

These tools allow you to navigate through your code, check variable values, and monitor the program's flow.

- Pros:
  - Provides full control over the execution flow.
  - You can inspect the entire program state at the point of the breakpoint.
  - Great for tracking down logical errors and understanding program behaviours step by step.
- Cons:
  - Requires knowledge of how to use the debugger effectively.
  - Sometimes difficult in multithreaded or asynchronous environments where many things happen simultaneously.

Core dumps

A core dump[1] is a file that captures the memory state of a program when it crashes. Core dumps have been used historically (and still are) to analyse what

[1]The term core dump originated from the era of core memory. A core dump refers to a snapshot of the contents of memory at a specific point in time, often when a program crashes or encounters a critical error. Initially, this was literally a "dump" of the contents of core memory to an external file or output device. The purpose was to capture the entire state of a running program—its variables, the contents of registers, and the memory it was using—so that developers could analyse what went wrong. Core memory was an early type of memory using magnetic rings to store bits, in early computers.

went wrong after a crash. The dumped memory includes the stack, heap, and CPU registers at the time of failure.

When a program crashes (due to segmentation faults, illegal instructions, etc.), the operating system can generate a core dump. This can then be loaded into a debugger like GDB to inspect the state of the program at the point of failure.

```
> gdb ./program core
```

- Pros:
  - Allows for post-mortem debugging: You can analyse the crash long after it happened.
  - Useful for hard-to-reproduce bugs (e.g., random crashes or memory corruption).
- Cons:
  - Core dumps can be large and difficult to interpret without a detailed understanding of the system's memory and state at the time of the crash.
  - Requires a debugger capable of loading and analysing core files.

Memory and hex dumps

Memory dumps and hex dumps provide raw output of a program's memory at a given time, and developers can manually analyse these dumps to inspect variables, buffers, and structures. A hex dump displays memory contents in hexadecimal format, sometimes alongside ASCII interpretations.

Programs like 'hexdump', 'od' (octal dump), or 'xxd' are used to output memory contents. On systems with no advanced debugging tools, developers would often dump memory into a 'file.bin' and manually inspect it for corruption or unexpected values.

```
> hexdump -C file.bin
```

- Pros:
  - Provides a very granular view of memory, useful for debugging low-level memory corruption or buffer overflows.
  - Works on systems without access to a sophisticated debugger.
- Cons:

  - Manually analysing memory in hexadecimal is time-consuming and error-prone.
  - Requires a strong understanding of memory layouts, offsets, and program internals.

Symbolic debugging

Symbolic debugging involves using debuggers that can map binary code back to high-level code constructs (like variable names, function names, etc.) using symbol tables. This became common with tools like 'GDB', 'dbx', and others.

When a program is compiled with debugging information (e.g., using the -g flag in GCC), the resulting binary contains *symbol information* that maps variables and functions to their addresses in the binary. A symbolic debugger can use this information to display variable values, function calls, and lines of code during debugging.

```
> gcc -g -o program program.c
```

- Pros:
  - Makes debugging much easier because the debugger can display high-level information rather than raw memory addresses or assembly.
  - Critical for debugging large, complex programs.
- Cons:
  - Larger binary size due to the inclusion of debug symbols.
  - Requires the code to be compiled with debug symbols, which may not always be possible (e.g., in third-party libraries).

Hardware debugging

Hardware debugging involves using specialised tools that interface directly with the microcontroller to inspect its internal state and control execution at the hardware level. For Raspberry Pi Pico and other microcontrollers, tools like the *Pico Debug Probe* and SWD (Serial Wire Debug) interface allow developers to halt execution, set breakpoints, and step through code in real-time, providing deep insight into the behaviour of embedded systems.

With the Pico Debug Probe, developers can monitor registers, memory, and peripherals during program execution. This hardware-based approach offers a higher level of control compared to software-only methods, making it invaluable when working with real-time systems or when precise timing is critical.

- Pros:
  - Direct access to hardware state, allowing for more precise and low-level debugging.
  - Essential for debugging timing issues, peripheral interaction, or hardware faults.
- Cons:
  - Requires additional hardware, such as a debug probe or SWD adapter. However, a second Pico can also be used as an alternative, functioning as a debug probe.
  - Can be more complex to set up compared to software-only debugging.

We will explore the Raspberry Pi Pico in the next Chapter 4, but for our purposes, we will rely on a UART connection instead of using the hardware debug option. This approach will suffice for the simple examples and educational focus of this material. While software-based debugging, like print statements or UART logging, provides basic insights into the program's flow, hardware tools such as the Pico Debug Probe offer more detailed control over elements like timing, performance, and real-time execution. This hardware-based option becomes especially valuable when working on larger, more complex Pico projects.

3.3.4 Summary

This section covers essential prerequisites, basic debugging techniques, and tools for programming on standard computers or microcontrollers. Readers are expected to have experience with C and Python. Debugging is broken down into syntax errors, logical errors, and runtime errors. Effective debugging includes systematically identifying reproducible issues, collecting data, forming and testing hypotheses, and applying solutions. Tools for debugging range from simple print statements and logging to more advanced methods like breakpoints, core dumps, memory dumps, and symbolic debugging, each offering unique advantages for different scenarios. Hardware debugging is also highlighted for its precision, especially in real-time systems, with tools like the Pico Debug Probe offering deep insights into hardware states valuable in embedded systems.

3.4 Optimisation

Optimising programs refers to the process of improving a program's performance in terms of *speed*, *memory usage*, or other resources like *disk* and *network I/O*. The goal is to make the program more efficient without changing its output or

behaviour. Program optimisation is essential in many contexts, from embedded systems where resources are limited, to high-performance computing where even small inefficiencies can be magnified.

Typical process

1. *Profiling.* Start by profiling your program to identify performance bottlenecks or hotspots.
2. *Targeting bottlenecks.* Focus your optimisation efforts on areas that contribute the most to performance degradation (e.g., slow loops, excessive memory usage, or I/O delays).
3. *Algorithmic improvements.* Once you've identified bottlenecks, explore whether a more efficient algorithm can replace the existing one.
4. *Fine-tuning with compiler optimisations, see Chapter 5.* Use compiler flags and optimisations to improve low-level performance. Profile-guided optimisations (PGO) can further enhance this.
5. *Memory and I/O optimisation.* After computational optimisations, focus on optimising memory access patterns and I/O operations, which may still be limiting performance.

Optimising programs is a multi-step process that involves analysing the performance of a program, identifying bottlenecks, and applying techniques ranging from compiler flags to algorithmic changes and parallelism. The right tools and techniques vary depending on the type of program and its use case. Profilers, memory analysers, and parallelisation tools are essential to this process, along with a deep understanding of the underlying hardware and algorithms.

Types

- *Compiler.* These are optimisations that happen during the compilation of source code to machine code. Compilers often have multiple levels of optimisation flags (like -O1, -O2, -O3 in GCC or Clang) that can help improve performance. Examples include:
  - Inlining: Replacing a function call with the body of the function to reduce overhead.
  - Loop unrolling: Expanding loops to reduce the number of iterations.
  - Dead code elimination: Removing code that is never executed.

  - Constant folding: Computing constant expressions at compile time instead of runtime.

- *Algorithmic.* This involves selecting more efficient algorithms for a given task. Often, significant improvements can be achieved by changing the algorithm, for example:

  - Replacing a bubble sort with quick sort or merge sort.
  - Using more efficient data structures like hash maps instead of lists for certain lookup operations.

- *Memory.* These optimisations reduce the memory footprint of an application, which can also lead to performance improvements. Examples include:

  - Memory pooling: Reducing the number of expensive memory allocation and deallocation operations.
  - Cache optimisation: Rearranging data structures to maximise cache locality, which reduces cache misses.
  - Compression: Using techniques like data compression to reduce memory usage at the expense of additional CPU overhead.

- *Parallelism and concurrency.* These optimisations involve taking advantage of multiple CPU cores, hardware threads, or distributed systems. Examples include:

  - Multithreading: Splitting a program into multiple threads to run on different cores.
  - Vectorisation: Leveraging SIMD (Single Instruction, Multiple Data) instructions to perform the same operation on multiple data points simultaneously.
  - GPU acceleration: Using Graphics Processing Units (GPUs) for tasks that can be parallelised, such as deep learning or graphics rendering.

- *I/O.* Many programs are bottlenecked by input/output operations, especially disk or network access. Optimising I/O can have a significant impact on performance:

  - Buffering: Using buffers to reduce the number of read/write operations.
  - Asynchronous I/O: Allowing the program to continue executing while waiting for I/O operations to complete.
  - Caching: Storing frequently accessed data in memory to avoid costly disk or network operations.

Tools

We will focus on some basic optimisation techniques below. Each tool you implement should serve a distinct purpose in enhancing the overall performance of a program or application. This will help you grasp how different optimisation methods target specific aspects of program efficiency.

1. *Profilers.* Profilers are tools that help developers understand the performance characteristics of their programs, pinpointing bottlenecks and inefficient code. Some well-known profilers include:
    - gprof: A profiling tool for Unix-based systems.
    - perf: A performance analysis tool available on Linux that provides various metrics like CPU cycles and cache misses.
    - Valgrind: A tool that detects memory issues, including leaks and invalid memory accesses.
    - Visual Studio: For .NET and Windows applications, and others, there are several profiler tools providing detailed performance metrics.
2. *Static analysers.* These tools analyse source code without running it, looking for potential optimisations and performance issues:
    - Clang Static Analyser: Provides warnings about inefficient code and potential bugs.
    - Coverity Scan: Another static analysis tool used for finding bugs and performance issues. (Can also be run on GitHub e.g.)
3. *Compiler tools.* Modern compilers come with optimisation capabilities built-in. For example:
    - GCC/Clang: Offers various levels of optimisations (-O1, -O2, -O3, -Ofast) and flags for specific types of optimisations.
    - LLVM: A framework for building custom optimisers, often used for more advanced or domain-specific optimisations.
    - Intel C++ Compiler: A compiler optimised for Intel hardware, providing advanced optimisations for vectorisation, parallelism, and CPU features.
4. *Memory optimisers.* These tools help analyse and reduce the memory usage of a program:
    - Massif: A heap profiler from Valgrind that helps understand memory consumption.

- dmalloc (debug malloc library): A C-based memory debugging and profiling tool.

Excluded from the above are e.g., parallelisation and I/O tools. Examples of the former is the well known CUDA, and of the latter IOzone.

Example: Profiling in steps

Profiling a C program involves measuring its performance to identify areas that may benefit from optimisation, such as slow-running functions or inefficient code paths. Here's an outline of how to profile a typical C program using common tools like 'gprof', which is a widely used profiler for Unix-like systems.

1. *Compile the program with profiling support.* First, you need to compile your C program with profiling enabled. This is typically done by passing the -pg flag to gcc, which instructs the compiler to include additional instrumentation in the program for profiling.

   ```
   > gcc -pg -o program program.c
   ```

   This compiles the 'program.c' file and produces an executable named 'program', with profiling support included.

2. *Run the program.* Next, you run the compiled program as usual. While the program runs, it collects profiling data about the function calls, how often each function is called, and the time spent in each function.

   ```
   > ./program
   ```

   After the program finishes executing, it generates a file called 'gmon.out' in the working directory, which contains the collected profiling data.

3. *Analyse the data.* To interpret the profiling data, you use 'gprof', which reads the 'gmon.out' file and generates a report. You can view this report by running (n.b. the first '>' is here an indicator of the prompt, only the second you enter):

   ```
   > gprof program gmon.out > profile.txt
   ```

   This command tells 'gprof' to analyse the executable 'program' and its associated 'gmon.out' file, and to output the results to a file called 'profile.txt'. You can open this file to review the profiling report.

4. *Interpret the report.* The profiling report generated by 'gprof' will include two main sections:

   - *Flat profile.* This shows how much time was spent in each function, both in terms of absolute time and as a percentage of the program's total runtime.
   - *Call graph.* This section provides detailed information about which functions are called which, how many times they were called, and how much time was spent in each function.

   Using this information, you can identify performance bottlenecks, such as functions that are called frequently or take up a large proportion of the program's execution time. You can then optimise those functions to improve performance.

   Example of a 'flat profile':

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 30.0      0.03     0.03      2000     0.01     0.02  functionA
 20.0      0.05     0.02      1500     0.01     0.03  functionB
 50.0      0.10     0.05      1000     0.05     0.05  functionC
```

5. *Optimise.* Next you follow the procedure from above. After identifying the bottlenecks, you can revise the code to optimise the problematic sections, recompile, and repeat the profiling process to assess the impact of the changes.

3.4.1 Memory

Memory optimisations aim to reduce the memory footprint of a program, which not only conserves system resources but also improves speed, particularly in memory-bound applications. Here are some common techniques:

1. Reducing memory usage

   - *Efficient data structures.* Selecting the right data structure is critical for memory optimisation. For example, replacing an array with a more compact structure like a bit vector can save significant memory when storing binary data. Using linked lists can be space-efficient when the dataset size is unpredictable, but contiguous arrays may be better when the size is fixed and known.

- *Smaller data types.* Avoid using larger data types where smaller ones suffice. For instance, if an integer only needs to store values between 0 and 255, use an 8-bit uint8_t instead of a 32-bit int.
- *Object pooling.* If your program frequently creates and destroys objects, you can use object pools to reuse objects rather than constantly allocating and deallocating memory. This minimises the overhead of memory allocation and reduces fragmentation.
- *Lazy initialisation.* Postpone the creation of objects or data until they are absolutely needed. By not initialising everything upfront, you can reduce the initial memory overhead.

2. Improving memory access patterns

   - *Cache locality.* CPUs rely heavily on caches (small, fast memory stores) to reduce the latency of memory access. Programs that access data in a predictable pattern (e.g., accessing elements in an array sequentially) tend to benefit from better cache locality. Structuring your data to allow for more cache-friendly accesses can significantly speed up execution.
   - *Temporal locality.* Reusing the same data within a short period.
   - *Spatial locality.* Accessing contiguous memory locations (as in array traversal).
   - *Avoiding memory fragmentation.* In long-running programs, continuous allocation and deallocation can lead to memory fragmentation, where free memory is split into small, non-contiguous chunks. This can be mitigated by:
     - Pre-allocating memory in chunks if possible.
     - Using memory allocators that minimise fragmentation (e.g., arena allocators or slab allocators).

3. Garbage collection tuning (in managed languages)

   For languages with automatic memory management (like Java, C#, or Python), optimising the garbage collector (GC) can have a significant impact on both memory and speed. (But it is not always possible to tune memory management from the programming language or virtual machine.)

   - *GC tuning.* Tuning garbage collection involves adjusting parameters such as heap size, GC intervals, and pause times. For example, a larger heap might reduce GC frequency but increase memory consumption, whereas a smaller heap results in more frequent GC cycles but conserves memory.

- *Minimising object creation.* In languages with garbage collection, reducing the creation of short-lived objects can reduce the load on the GC and prevent memory leaks.

4. Memory pools

   Using memory pools (also called memory arenas) for allocating memory in bulk can reduce fragmentation. This is especially useful in embedded systems or applications with constrained memory. Memory pools allocate large blocks of memory and hand out chunks from this pre-allocated pool, speeding up allocation/deallocation since these operations don't involve the system's general-purpose allocator.

3.4.2 Time

Speed optimisations aim to reduce the time a program takes to execute by making it more computationally efficient or by reducing bottlenecks in processing.

1. Algorithmic optimisation

   Time complexity. One of the most significant factors in speed optimisation is choosing algorithms with better time complexity. For example, an $O(n \log n)$ sorting algorithm (like quicksort in best case) is significantly faster than $O(n^2)$ algorithms (like bubble sort) for large datasets.[2]

   Avoiding redundant computations. Reusing previously computed results instead of recalculating them can significantly improve performance. Techniques like memoization (storing the results of expensive function calls) or tabulation (building up a table iteratively) help reduce redundant operations, particularly in recursive algorithms. Additionally, dynamic programming is a powerful approach to optimise such problems by solving subproblems once and reusing their results.

2. Loop optimisations

   Loop unrolling. Manually or automatically (via compiler optimisations, see Chapter 5) expanding loops so that multiple iterations are executed per cycle. This reduces the loop control overhead (like incrementing a counter or checking the loop condition). (Compare with the possibilities for unrolling in the Fibonacci series example, as discussed in Section 2.3.)

[2] Big O notation is a way in computer science to describe how an algorithm's time or space needs grow as the input size gets larger. It helps compare algorithms by focusing on their efficiency, especially in the worst case, without worrying about small details that don't affect overall growth.

Reducing loop nesting. Minimising nested loops can have a dramatic effect on performance since the complexity increases with the depth of nesting. Where possible, flattening loops or breaking complex nested operations into simpler forms can improve speed.

Minimising expensive operations in loops. Move calculations or conditions outside the loop if they don't need to be recalculated each iteration. For instance, calculating something like 'array.length' repeatedly in a loop is inefficient if the length doesn't change. (However this might also depend on e.g., compilers that are smart enough to optimise this.)

3. Concurrency and parallelism

   Multithreading Taking advantage of multiple CPU cores by running different parts of the program in parallel can lead to massive performance improvements. Libraries like OpenMP or pthreads in C/C++ or thread libraries in Python and Java allow you to parallelise tasks.

   Asynchronous programming. Non-blocking I/O operations allow a program to perform useful work while waiting for I/O tasks to complete. Asynchronous frameworks (e.g., Python's 'asyncio') or event-driven models (e.g., 'Node.js') allow for concurrent I/O, significantly improving responsiveness in I/O-bound applications.

4. Minimising I/O bottlenecks

   Batching I/O. Instead of handling I/O operations individually (which can be expensive), batch multiple operations together. For example, writing data to a file in larger chunks rather than frequently writing small amounts can improve performance.

   Disk and network caching. Caching frequently used data in memory reduces the number of slow disk or network accesses.

Combining memory and speed optimisations

Memory-speed trade-offs. Often, memory and speed optimisations require a balance. For example, using larger data structures (e.g., precomputed lookup tables) might increase memory usage but reduce computation time, leading to faster execution. Conversely, using smaller, more compact data structures may conserve memory but result in slower processing due to increased computation or less cache efficiency.

Data locality. Good memory management often enhances speed by reducing cache misses. Rearranging data structures to ensure that frequently accessed data is close together in memory can lead to significant speedups by reducing memory latency.

3.4.3 Dynamic Programming

Dynamic programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems and solving each subproblem only once. Instead of recalculating solutions multiple times, DP stores results and reuses them, significantly improving efficiency. A problem is suited for DP if it has:

- *Optimal Substructure.* The optimal solution can be built from the optimal solutions of subproblems.
- *Overlapping Subproblems.* The same subproblems are solved multiple times.

Recursive approach solves problems by breaking them into smaller ones but with *redundant* calculations.

- *Memoization (Top-Down DP)* solves problems recursively but stores results in a table to avoid recomputation.
- *Tabulation (Bottom-Up DP)* solves the smallest subproblems first and builds up to the final solution iteratively.

Example: Fibonacci Sequence

The Fibonacci sequence follows the relation:

$$F(n) = F(n-1) + F(n-2)$$

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}
```

The naïve recursive approach with an exponential complexity has a time complexity of $O(2^n)$ due to repeated calculations.

But if we instead of recomputing values, store them in an array:

```
#include <stdio.h>
#define MAX 100

int fibMemo[MAX];

int fibonacci(int n) {
    if (n <= 1)
        return n;
```

```
    if (fibMemo[n] != -1)
        return fibMemo[n];
    fibMemo[n] = fibonacci(n - 1) + fibonacci(n - 2);
    return fibMemo[n];
}

int main() {
    int n = 10;
    for (int i = 0; i < MAX; i++)
        fibMemo[i] = -1; // init array with -1
    printf("Fibonacci(%d) = %d\n", n, fibonacci(n));
    return 0;
}
```

This is a variant of Memoization (Top-Down DP) for computed values that are stored in `fibMemo[]` to avoid redundant calls.

A third option is instead of recursion, we build solutions iteratively:

```
int fib(int n) {
    int dp[n+1];
    dp[0] = 0; dp[1] = 1;

    for (int i = 2; i <= n; i++)
        dp[i] = dp[i-1] + dp[i-2];

    return dp[n];
}
```

This eliminates recursion, making it more efficient, and uses what is called Tabulation (Bottom-Up DP).

Example: 0/1 Knapsack Problem

The 0/1 Knapsack problem is a classic optimisation problem where a set of items, each with a given weight and value, must be selected to maximise the total value while ensuring that the total weight does not exceed a given limit. Unlike the *fractional knapsack problem*, where items can be divided, the 0/1 knapsack problem requires each item to be either included in full or excluded entirely. This makes it a combinatorial problem well-suited for dynamic programming, as the optimal solution for a given weight limit depends on the optimal solutions for smaller weight limits.

Given N items with weights $wt[]$ and values $val[]$, and a knapsack with weight limit W, maximise the total value.

Recursive Approach (Exponential Complexity). Tries all combinations of including or excluding an item.

DP Approach (Polynomial Complexity). Define $dp[i][w]$ as the maximum value that can be obtained using the first i items and weight limit w.

$$dp[i][w] = \max(dp[i-1][w], val[i-1] + dp[i-1][w - wt[i-1]])$$

When to Use Dynamic Programming

Dynamic programming is useful in problems that exhibit both optimal substructure and overlapping subproblems. If a brute-force recursive approach leads to redundant computations and inefficiency, DP can help by storing and reusing previous results. It is particularly effective when a problem can be broken into stages, where solving smaller subproblems contributes directly to constructing the final solution.

3.4.4 Confusion matrix

Machine learning, a subfield of artificial intelligence (AI), focuses on creating algorithms and models that learn patterns from data to make predictions or decisions without explicit programming for specific tasks. For example, instead of writing hard-coded rules to classify emails as "spam" or "not spam," a machine learning algorithm identifies patterns in labeled examples and generalises them to unseen emails.

Teaching machine learning early equips you with tools to analyse complex systems, make data-driven decisions, and understand modern technologies. Beyond building models, a critical skill in machine learning is the ability to evaluate and refine them. This brings us to the concept of debugging in machine learning, which is inherently different from traditional software debugging.

In traditional software development, debugging typically involves identifying and fixing specific coding errors or logic flaws—answering questions like "how did this error occur?" Debugging machine learning models, however, focuses on understanding why a model's performance is suboptimal. This might include questions like:

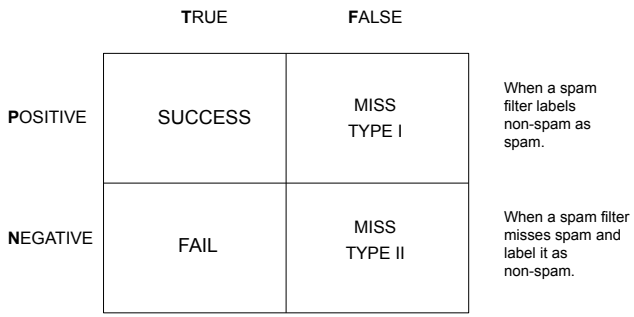
- Why is the model misclassifying certain data points?
- Why does it underperform on specific subsets of the data?

These questions arise because machine learning models derive their behaviour from complex interactions among data patterns, model architectures, and features. Debugging in this context involves examining data distributions, interpreting

model decisions, and identifying root causes of errors. This makes machine learning debugging as much about insight and understanding as it is about resolving specific bugs.

A confusion matrix is an essential diagnostic tool in machine learning that provides a detailed breakdown of a classification model's predictions. For binary classification, it is structured as follows:

- **True Positive (TP)**: The model correctly predicts the positive class (e.g., a spam filter correctly identifies spam).
- **True Negative (TN)**: The model correctly predicts the negative class (e.g., a spam filter correctly identifies non-spam).
- **False Positive (FP)**: The model incorrectly predicts the positive class (e.g., a spam filter labels non-spam as spam). This is also called a *Type I error*.
- **False Negative (FN)**: The model incorrectly predicts the negative class (e.g., a spam filter misses spam and labels it as non-spam). This is a *Type II error*.



A *confusion matrix* is commonly used in machine learning to summarise these values for an entire dataset. From this, various performance metrics can be derived, such as precision, recall, and F1-score. These metrics allow developers to fine-tune and troubleshoot model accuracy and the balance between sensitivity (true positive rate) and specificity (true negative rate).

Model debugging

In debugging and optimisation, a confusion matrix helps identify weaknesses in a model. For example, if the false positive rate is high, the model might be "over-calling" the positive class, which could be mitigated by adjusting the decision

threshold. Alternatively, if false negatives are high, the model might be missing important cases, prompting a closer look at data features or model complexity.

The confusion matrix is therefore critical for diagnosing and improving model reliability and for understanding the types of errors the model is most likely to make, both of which are essential for fine-tuning algorithms in real-world applications.

The confusion matrix is a valuable debugging tool because it reveals the types of errors a model is making, which can guide developers in refining and adjusting the model. When debugging a model, analysing each component of the confusion matrix—true positives, true negatives, false positives, and false negatives—can highlight where the model is strong and where it needs improvement. Here's how this works in practice:

High False Positives (FP). When the model has a high rate of false positives, it means it frequently predicts the positive class when it shouldn't. For example, in a spam filter, high false positives mean non-spam emails are being marked as spam. This issue could frustrate users, as important emails may be missed. To debug this, developers might:

- Adjust the decision threshold for the positive class, making it more conservative.
- Re-evaluate feature importance to check if certain features are overly influencing the positive predictions.
- Inspect data balance to see if the training set has more positive cases, leading the model to over-predict this class.
- Consider additional features or fine-tune existing ones that could help differentiate between positive and negative classes.

High False Negatives (FN). A high rate of false negatives indicates that the model often fails to detect the positive class. In a medical diagnosis model, for instance, high false negatives mean the model is missing cases of disease, which could have severe consequences. To debug this, developers might:

- Lower the decision threshold for classifying a positive case, making the model more sensitive.
- Explore feature engineering to see if additional informative features could help identify positives more accurately.
- Analyse data quality to ensure important features are well-represented in the training data.

- Use data augmentation techniques if the positive class is underrepresented, which could help the model learn to identify these cases better.

Examining True Positives (TP) and True Negatives (TN). While true positives and true negatives indicate correct predictions, it can be useful to review these instances as well. For example:

- Review True Positives: Checking the features in true positives can help confirm whether the model is using relevant information correctly, giving insight into whether similar features could improve predictions for borderline cases.
- Review True Negatives: Similarly, understanding what makes a true negative may help refine the model to avoid false positives.

Using the confusion matrix to guide model tuning. Debugging often involves iterating over different aspects of the model, and the confusion matrix helps focus these efforts. For instance, depending on whether the model's goal is higher precision (fewer false positives) or higher recall (fewer false negatives), developers can use the confusion matrix to tune hyperparameters or decision thresholds accordingly.

Incorporating feedback loops. As part of a debugging and improvement cycle, the confusion matrix is often examined over multiple stages of training and deployment. In real-world applications, user feedback (e.g., users marking emails as "not spam" when falsely classified) can be fed back into the training process. The model can then be updated to reduce the specific error rate, such as lowering false positives in a spam filter.

The confusion matrix is like a 'map of model errors,' allowing developers to "zoom in" on specific issues in performance. By systematically analysing each quadrant, developers can identify not only where the model is failing but also use these insights to make targeted adjustments to the training process, feature set, or decision-making strategy of the model. This makes the confusion matrix an essential tool for debugging and iterative improvement in model development.

Example: Temperatures

Assume we have time series data of temperature measurements along with corresponding temperature predictions for each time point. By correlating these actual and predicted temperature pairs, we can evaluate and refine our prediction model to improve its accuracy. Using a confusion matrix, we can systematically identify patterns in the model's errors, allowing us to debug and adjust the model for better alignment with the actual temperature trends. Let's illustrate this with an example in Python.

1. Define accuracy thresholds: Since temperatures are continuous values, we need a threshold to classify each prediction as a "hit" or "miss." For example, you could consider a prediction "accurate" if it's within +/-2 degrees of the actual temperature. This is naturally an assumption that can be changed.
2. Classify each prediction: Loop through each actual and predicted temperature pair in the log and classify them based on the defined threshold(s).
3. Count hits and misses: Use these classifications to populate a confusion matrix. For simplicity, we can start with a binary confusion matrix with categories like "accurate" (True Positive and True Negative) and "inaccurate" (False Positive and False Negative).

```
# sample: each element is (actual temperature, predicted temperature)
temperature_log = [
    (20, 21), (25, 24), (30, 28), (15, 18), (22, 19), (27, 29),
    (18, 17), (21, 23), (16, 15), (20, 20), (24, 26), (19, 18),
    (23, 24), (28, 27), (19, 20), (17, 18), (22, 23), (26, 25),
    (29, 31), (21, 22), (18, 16), (24, 22), (20, 21), (25, 27)
]

# def. threshold for prediction as "accurate"
threshold = 2 # plus or minus 2 degrees range

true_positive = 0 # accurate positive predictions
false_positive = 0 # predicted higher, inaccurate
true_negative = 0 # accurate negative predictions
false_negative = 0 # predicted lower, inaccurate

# populate confusion matrix
for actual, predicted in temperature_log:
    difference = abs(actual - predicted)

    if difference <= threshold:
        # within acceptable range
        if predicted >= actual:
            true_positive += 1 # slightly higher or equal
        else:
            true_negative += 1 # slightly lower
    else:
        # outside acceptable range
        if predicted > actual:
            false_positive += 1 # overestimated and inaccurate
        else:
            false_negative += 1 # underestimated and inaccurate
```

```
print("Confusion matrix")
print(f"True Positives (TP): {true_positive}")
print(f"False Positives (FP): {false_positive}")
print(f"True Negatives (TN): {true_negative}")
print(f"False Negatives (FN): {false_negative}")

# optional: calculate metrics
accuracy = (true_positive + true_negative) / len(temperature_log)
precision = true_positive / (true_positive + false_positive) if (
    true_positive + false_positive) > 0 else 0
recall = true_positive / (true_positive + false_negative) if (
    true_positive + false_negative) > 0 else 0

print("\nPerformance metrics")
print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
```

We define an error threshold (e.g., +/-2 degrees) for what counts as an accurate prediction. This threshold can be adjusted based on the acceptable margin of error. For each pair in the log, the program calculates the absolute difference between the actual and predicted temperatures. If the difference is within the threshold, the prediction is counted as "accurate"; otherwise, it's "inaccurate." We then assign these to the true/false positive/negative categories based on whether the prediction was above or below the actual temperature.

Metrics

Recall may be the lesser known metric. In the context of a confusion matrix, recall is a metric that measures a model's ability to identify all relevant instances of the positive class. It is also known as the true positive rate or sensitivity. Recall tells us the proportion of actual positive cases that the model correctly identifies as positive. It is calculated as the ratio of true positives (TP) to the sum of true positives and false negatives (FN).

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

High recall indicates that the model successfully captures most of the positive instances, which is crucial in applications like disease screening, where missing positives can be costly.

Precision measures the proportion of predicted positive cases that are actually correct. It is the ratio of true positives to the sum of true positives and false

positives (FP).

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

High precision indicates that when the model predicts a positive, it is likely to be correct. Precision is essential in applications where false positives are undesirable, such as in spam filtering.

Accuracy represents the overall correctness of the model, measuring the proportion of all correct predictions (true positives and true negatives) out of the total predictions made.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

Accuracy is a general measure of model performance, but can be misleading if the classes are imbalanced.

The unimplemented F1 score is a metric that combines precision and recall into a single value, offering a balance between the two.

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

3.4.5 Summary

This section explores optimisation and debugging techniques in programming, with a focus on enhancing performance, memory usage, and algorithm efficiency. Key strategies include profiling to identify bottlenecks, applying algorithmic and memory optimisations, and leveraging compiler settings for efficiency. Debugging is categorised by error types—syntax, logical, and runtime—with tools such as print statements, logging, breakpoints, and memory analysis. Dynamic programming is an optimisation technique used to solve problems by breaking them into overlapping subproblems, storing their solutions, and reusing them to avoid redundant computations. For machine learning models, this section exemplify using confusion matrices to diagnose performance issues, detailing how metrics like precision, recall, and accuracy help refine model predictions and guide tuning for improved reliability.

3.5 Tests and testing

Testing is an important aspect of software development that ensures code behaves as expected under various conditions and meets the specified requirements.

It helps identify and address defects before the software is deployed, thereby enhancing the reliability and quality of the final product.

Code coverage is a metric in software testing that measures how much of the codebase is exercised by the tests. It highlights which parts of the code are covered by tests and, more importantly, which parts are not. High code coverage can indicate that the software is thoroughly tested, though it does not guarantee the absence of bugs. Tools like 'gcov' (for C/C++), 'coverage.py' (for Python), or integrated features in CI pipelines can be used to analyse coverage, providing developers with reports that guide test improvement efforts.

There are several types of tests, each with a specific focus and purpose. Here's a brief overview of some types of tests used in development, along with benefits and how they work. As can be noted, tools such as 'assert' in Python can be used in various types of tests and are not associated with only one type.

Unit tests

- Purpose: To verify that individual units of code (e.g., functions, methods, classes) work correctly in isolation.
- Why: Catching bugs early in the development process and ensuring that each small piece of code behaves as expected.
- How: Each unit test focuses on a single function or method, providing known inputs and checking the outputs against expected values. For example, testing a function that adds two numbers would ensure that given 2 and 3, it returns 5.

```
def test_add():
    assert add(2, 3) == 5
```

When developers modify code—whether through adding new features, refactoring, or fixing bugs—there's a risk that the changes could unintentionally affect parts of the system that were previously functioning as expected. A regression happens when a modification introduces errors or failures in those unaffected areas.

Unit tests are often automated and run frequently to detect these regressions early. If a unit test that previously passed starts failing after a change, it signals that a regression has occurred, and the developers can investigate and fix the issue before it becomes a bigger problem.

Integration tests

- Purpose: To ensure that different units or modules work together correctly.

- Why: Verifying interactions between components and making sure integrated systems function as expected.
- How: These tests check whether multiple components (like database access and APIs) work together in real-world scenarios. For example, an integration test might check if a web service correctly retrieves data from a database and formats it for display.

```
    def test_service_integration():
        response = get_data_from_service()
        assert response.status_code == 200
```

Integration tests run after unit tests to ensure the correct behaviour when components are combined.

Functional tests

- Purpose: To validate the software against functional requirements and ensure it behaves according to the specifications.
- Why: Testing whether specific features work as expected from the end-user's perspective.
- How: Functional tests focus on verifying that specific functionalities (e.g., user login, form submission) behave correctly. They are often higher-level than unit tests and can simulate real user actions.

```
def test_user_login():
    assert login(username="test", password="correct") == "Login
        successful"
```

These tests ensure that the software meets its functional requirements.

Regression tests

- Purpose: To ensure that new changes or updates to the codebase do not introduce bugs or break existing functionality.
- Why: Maintaining stability and catching issues introduced by recent changes.
- How: Regression tests are a *collection of previously written tests* (often unit, integration, or functional tests) that are rerun after updates. If a test that previously passed now fails, the developer knows the recent changes caused a problem.

Example: If a new feature was added to a project, regression tests would check whether the existing features continue to work as expected after the change.

Smoke Tests

- Purpose: To perform a quick check that the most critical parts of the software work correctly.
- Why: Running quick validations before a more thorough testing phase.
- How: Smoke tests are a subset of tests that ensure that the core functionality of the application is operational. They are often run after a build to confirm that the system is stable enough for further testing.

Example: A smoke test for a web app might ensure that the application loads and that users can log in without errors.

Performance Tests

- Purpose: To evaluate the software's performance under specific conditions (e.g., load, stress).
- Why: Ensuring that the system performs efficiently and remains responsive under high usage.
- How: Performance tests measure various aspects like response time, throughput, and resource usage. This may include load testing (to check how the system handles multiple users) and stress testing (to evaluate the system's behaviour under extreme conditions).

Example: A performance test might simulate 10,000 users accessing an API at the same time and measure the response time or memory consumption.

Security Tests

- Purpose: To ensure that the application is free from vulnerabilities and that sensitive data is protected.
- Why: Identifying security flaws such as SQL injection, cross-site scripting (XSS), or unauthorised access.
- How: Security tests simulate attacks on the system to expose weaknesses. These can be automated or using manual penetration tests.

Example: A security test might try to inject malicious SQL code into a login form to check if the system is vulnerable.

Property-Based Tests

- Purpose: To test that the software satisfies general properties or invariants over a wide range of inputs.
- Why: Discovering edge cases or unexpected input behaviour that may not be covered by example-based unit tests.
- How: Instead of testing with specific inputs, property-based testing generates many randomised or structured test cases and verifies that a certain property always holds. This is useful for functions where algebraic laws or consistency conditions can be stated. Tools like Hypothesis (Python), QuickCheck (Haskell), or ScalaCheck (Scala) are commonly used.

Example: Testing that a sorting function always returns a list in non-decreasing order, regardless of input.

```
from hypothesis import given
import hypothesis.strategies as st

@given(st.lists(st.integers()))
def test_sort_order(xs):
    assert sorted(xs) == sorted(xs)
```

In a later chapter you will find a separate Section 8.10 on Property-Based Testing.

Fuzz Tests

- Purpose: To detect bugs, crashes, or security issues by feeding programs with large amounts of random or malformed input.
- Why: Revealing unexpected behaviour that might arise under rare or adversarial conditions.
- How: Fuzzing tools automatically generate and inject a vast number of inputs into the system to explore uncommon paths. This is especially useful in systems that parse complex input formats, such as file decoders, network services, or compilers. Popular fuzzers include AFL (American Fuzzy Lop), libFuzzer, and fuzzing support in tools like clang or Python's 'atheris'.

Example: Fuzzing a function that parses JSON documents to discover inputs that cause parsing errors or crashes.

Summary of testing types

- Unit: Validate individual components in isolation. Very granular, tests small pieces of code. Often used after each build.
- Integration: Ensure components work together correctly. Tests the interaction between various system components. Used after adding or modifying major components.
- Regression: Ensure that new changes don't break existing functionality. Re-running previous tests on affected areas. After code changes, used before releases.
- Smoke: Quickly validate critical parts of the system after a build. Essential, high-level functionality check. Often applied after each build.
- Performance: Check system performance under load and stress. Tests system performance under various conditions (e.g., load, stress). Run before major releases or updates.
- Security: Identify and mitigate security vulnerabilities. Tests for vulnerabilities, such as SQL injection, or cross-site scripting (XSS). Regularly runs, especially before a public release.
- Functional: Verify specific functions or features work as expected. Tests end-to-end business processes. After adding or modifying features, checking.
- Property-based: Test general properties rather than specific inputs/outputs. Automatically generates a wide range of inputs to uncover edge cases. Useful for discovering unexpected behaviours or assumptions. (See Section 8.10.)
- Fuzz: Provide random, unexpected, or malformed input to the program. Effective for security and robustness testing by identifying crashes and vulnerabilities. Especially valuable for input-handling code.

Higher level tests slightly more user oriented and excluded from above are e.g., End-to-End (E2E), usability or acceptability tests.

3.5.1 Automated testing and continuous integration

Automated tests are a key component in ensuring software stability as code is modified, enhanced, or refactored. These tests—whether they are unit tests, integration tests, or functional tests—are designed to run automatically, often in response to code changes. By automating tests, developers gain several advantages:

1. Immediate feedback: When tests fail after a change, developers are alerted right away, helping to catch and fix issues early.
2. Regression detection: Automated tests prevent regressions by verifying that new code doesn't break existing functionality.
3. Faster development: Automated tests allow for rapid, iterative development, freeing developers from manual testing and allowing them to focus on feature development.

Using make for test automation

The 'make' tool is often used to automate test execution, especially in C/C++ development but also in many other environments. make allows developers to define rules in a 'Makefile' that specify how to build, test, and deploy code. With make, we can create a standardised set of commands for compiling code, running tests, and generating reports, which can then be executed with a simple command.

For example, here's a basic 'Makefile' that includes targets for running unit tests:

```
# Makefile

# compile code
compile:
    gcc -o my_program main.c

# run tests
test: compile
    ./run_tests.sh

# cleaning up generated files
clean:
    rm -f my_program
```

In this example 'make compile' compiles the program. The 'make test' first compiles the code and then runs a script to execute tests. And 'make clean' removes any files generated during compilation, keeping the environment clean.

By running 'make test,' a developer can execute all tests, which is especially helpful when testing frequently.

Continuous Integration (CI) pipelines

Continuous Integration (CI) pipelines are systems that automatically run tests and other checks each time code is committed to the repository. Tools like GitHub

Actions, GitLab CI, Jenkins, and CircleCI are commonly used for setting up CI pipelines. A typical CI pipeline includes stages such as:

1. Building the code: Compiling or setting up dependencies to ensure code can be built from scratch.
2. Running tests: Executing unit tests, integration tests, and sometimes functional tests.
3. Static analysis: Checking code for style, quality, and common errors using tools like linters.
4. Deployment (optional): Pushing changes to staging or production environments if all tests pass.

Here's a simplified example of a CI pipeline configuration in GitHub Actions:

```
# .github/workflows/ci.yml

name: CI Pipeline

on:
  push:
    branches:
      - main
  pull_request:

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - name: Check out repository
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.8'

      - name: Install dependencies
        run: pip install -r requirements.txt

      - name: Run unit tests
        run: pytest tests/
```

In this configuration:

- The CI pipeline triggers on every push to the main branch and on pull requests.
- It sets up the environment, installs dependencies, and runs all tests in the tests/ directory.

This CI pipeline ensures that tests run on every code change, immediately notifying the team if any test fails.

Unit testing automation

By automating unit tests, developers can rapidly verify that the foundational parts of the code are working correctly, providing confidence when changes are made.

Tools like Pytest for Python, JUnit for Java, and Google Test for C++ facilitate unit testing by providing frameworks to:

- Write test cases in a consistent, organised manner.
- Run all tests automatically and generate reports on the results.
- Use mocking and stubbing for testing in isolation by simulating dependencies.

For instance, in Python with Pytest, a unit test might look like this:

```
def add(a, b):
    return a + b

def test_add():
    assert add(3, 4) == 7
    assert add(0, 0) == 0
    assert add(-1, 1) == 0
```

Combined process

By using tools like make, CI pipelines, and automated unit tests, we can create a robust and continuous validation process. Here's how these elements interact:

1. Developers write code and create *unit tests* to verify each part of their code.
2. Automated unit tests can be run locally with make test (or a similar command), ensuring new code functions as intended.

3. When code is committed to the repository, the *CI pipeline* triggers, automatically compiling the code, running tests, and generating feedback for the team.

4. If all tests pass, the pipeline may proceed to *deployment*; if any tests fail, developers receive feedback to address the issues immediately.

This process enhances software quality, speeds up development, and reduces the likelihood of bugs reaching production, making automated testing an essential part of the development workflow.

A Custom Test Virtual Machine

As you might have noticed, VMs are a natural way of approaching code and computers in this book. An approach to unifying testing across different types and languages is to execute tests within a dedicated *Test Virtual Machine* (Test VM). In this model, tests are expressed not as direct code in a specific programming language, but as a sequence of instructions in a language-neutral *intermediate representation* (IR). This IR is then executed by the Test VM in a controlled and deterministic environment.

The concept is analogous to how a general-purpose virtual machine executes compiled bytecode: the Test VM is provided with a minimal instruction set for test execution, such as loading input data, invoking functions or operations, checking assertions, logging results, and controlling timing. Each supported programming language or test framework can emit its own tests in this common IR format, enabling consistent execution and reporting regardless of the source language.

- *Language independence.* Tests written in Python, C, Java, or perhaps more plausible other custom languages can be compiled to the same IR and executed identically.

- *Reproducibility.* The Test VM provides deterministic execution, ensuring that tests behave the same way across platforms.

- *Extensibility.* The instruction set can be extended to support specialised testing domains, such as hardware integration or performance measurement.

- *Unified reporting.* Test results can be collected in a single, standardised format, simplifying integration with continuous integration (CI) systems.

Example IR: A simple test for an `add` function might be represented as:

```
LOAD_INPUT int 2
LOAD_INPUT int 3
CALL_FUNC add
ASSERT_EQ result 5
```

Here, `LOAD_INPUT` pushes input values into the test context, `CALL_FUNC` invokes the target operation, and `ASSERT_EQ` checks the result. This IR could be generated from unit tests, integration tests, or even property-based tests, all of which can be run by the same Test VM.

Integration in development workflows. The Test VM can be integrated into automated pipelines by providing a single executable that takes compiled test IR files as input and produces a unified test report. This means that developers can write tests in their preferred language or framework, while the execution, logging, and result aggregation are handled centrally. Such a system can also be extended to run tests in simulation before deployment to production or hardware targets, increasing safety and confidence in changes.

3.5.2 Summary

This section highlights the importance of various software testing types—unit, integration, functional, regression, smoke, performance, and security tests—in ensuring code functions as intended and meets requirements. Unit tests verify individual components, while integration and functional tests check the interaction between parts and overall system behaviour from the user's perspective. Regression tests ensure changes don't break existing functionality, and smoke tests provide quick checks on core features. Performance and security tests assess system efficiency and identify vulnerabilities. Automated testing, often managed with tools like make and CI pipelines, ensures continuous validation, providing immediate feedback on code changes, detecting regressions, and speeding up development, all of which contribute to improved software quality and reduced risk of bugs.

3.6 Advanced Debugging Practices

The debugging techniques covered in Section 3.3 focus primarily on development-time scenarios where developers have direct access to source code, compilation tools, and controlled environments. However, modern software systems present additional debugging challenges that require specialised approaches. This section explores advanced debugging practices essential for robust software development, covering error handling strategies that prevent bugs before they occur and techniques for debugging systems in production environments.

3.6.1 Error Handling and Resilience

Effective error handling serves as both a debugging prevention mechanism and a debugging aid. Rather than waiting for failures to occur, defensive programming practices anticipate potential issues and provide mechanisms for graceful handling when problems arise. The philosophy underlying these practices is that software should fail safely and provide clear diagnostic information when failures are unavoidable.

Defensive Programming

Defensive programming involves writing code that anticipates and handles unexpected conditions, invalid inputs, and edge cases. This approach reduces the likelihood of bugs reaching production and provides better diagnostic information when issues do occur. The core principle is to assume that anything that can go wrong will go wrong, and to prepare the code accordingly.

The fundamental strategy of defensive programming is to validate all assumptions explicitly rather than relying on implicit contracts. This means checking inputs, verifying preconditions, and ensuring that the program state remains consistent even when unexpected conditions arise.

Input validation represents the first line of defence against many categories of bugs. By checking parameters before using them, functions can avoid undefined behaviour and provide meaningful error messages instead of cryptic crashes:

```
int divide(int numerator, int denominator) {
    // Defensive check prevents division by zero
    if (denominator == 0) {
        fprintf(stderr, "Error: Division by zero attempted\n");
        return -1; // Or use error code/exception
    }

    return numerator / denominator;
}
```

This simple validation prevents a common runtime error while providing immediate feedback about the problem. The error message clearly indicates what went wrong, making debugging significantly easier than dealing with a generic arithmetic exception.

Bounds checking prevents buffer overflows and array access violations, which are among the most dangerous and difficult-to-debug errors in systems program-

ming. These checks transform potentially catastrophic memory corruption into controlled, debuggable error conditions:

```
void safe_array_access(int *array, size_t array_size, size_t index) {
    if (array == NULL) {
        fprintf(stderr, "Error: Null array pointer\n");
        return;
    }

    if (index >= array_size) {
        fprintf(stderr, "Error: Index %zu out of bounds (size: %zu)\n"
             , index, array_size);
        return;
    }

    // Safe to access array[index]
    printf("Value at index %zu: %d\n", index, array[index]);
}
```

The bounds checking here serves multiple debugging purposes: it prevents memory corruption that could manifest as mysterious bugs elsewhere in the program, provides precise information about the invalid access attempt, and maintains program stability even when logic errors occur.

Resource management ensures proper cleanup and prevents memory leaks, which can be particularly challenging to debug in long-running programs. Defensive resource management makes resource lifecycle explicit and provides clear error reporting when resource operations fail:

```
FILE* safe_file_operation(const char* filename) {
    if (filename == NULL) {
        fprintf(stderr, "Error: Null filename provided\n");
        return NULL;
    }

    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        fprintf(stderr, "Error: Cannot open file '%s': %s\n",
                filename, strerror(errno));
        return NULL;
    }

    return file; // Caller responsible for fclose()
}
```

This approach transforms a potential silent failure into an explicit, debuggable condition. The error message includes both the attempted operation and the system-provided reason for failure, giving developers the information needed to understand and fix the problem.

Exception Handling Patterns

In languages supporting exceptions, structured error handling provides mechanisms for separating error-handling code from normal program flow, making both easier to understand and debug. Even in C, similar patterns can be implemented using error codes and goto statements for cleanup. The key insight is that error handling should be systematic and predictable, not ad-hoc.

Exception handling patterns establish clear contracts about how errors propagate through the system. This predictability is crucial for debugging because it means that error conditions follow well-defined paths that can be traced and understood.

Python exception handling example:

```
def process_data_file(filename):
    try:
        with open(filename, 'r') as file:
            data = file.read()
            result = parse_data(data)
            return result
    except FileNotFoundError:
        logging.error(f"File not found: {filename}")
        raise # Re-raise for caller to handle
    except PermissionError:
        logging.error(f"Permission denied: {filename}")
        return None
    except Exception as e:
        logging.error(f"Unexpected error processing {filename}: {e}")
        # Log full traceback for debugging
        logging.debug("Full traceback:", exc_info=True)
        return None
```

This pattern demonstrates several important debugging principles. First, specific exceptions are handled differently, providing targeted error messages that help identify the root cause. Second, unexpected exceptions are caught and logged with full traceback information, ensuring that debugging information is preserved even for unanticipated errors. Third, the exception handling strategy is explicit about what should be recovered locally versus what should be propagated

to callers.

Error propagation strategies determine how errors should be communicated up the call stack. The choice of strategy significantly affects debugging experience because it determines where and how error information becomes available:

- *Fail fast.* Immediately terminate on critical errors to prevent data corruption. This strategy prioritises system integrity over availability and makes debugging easier by stopping execution at the point of failure rather than allowing the error to propagate and cause secondary problems.
- *Graceful degradation.* Provide reduced functionality when non-critical components fail. This approach maintains system availability but requires careful logging to ensure that degraded conditions are visible to debugging efforts.
- *Retry mechanisms.* Attempt recovery from transient failures. Retry logic must be carefully instrumented to distinguish between transient issues that resolve themselves and persistent problems that require investigation.
- *Circuit breakers.* Prevent cascading failures in distributed systems. Circuit breakers provide debugging benefits by isolating failing components and providing clear metrics about failure patterns.

Assertions and Contracts

Assertions serve as both documentation and debugging aids, explicitly stating assumptions about program state. They help catch logic errors during development and provide diagnostic information when assumptions are violated. The power of assertions lies in their ability to make implicit assumptions explicit and to fail immediately when those assumptions are violated, rather than allowing the program to continue in an inconsistent state.

Assertions are particularly valuable because they document the programmer's intent directly in the code. When an assertion fails, it indicates exactly which assumption was violated, providing a clear starting point for debugging investigations.

```
#include <assert.h>

void binary_search(int *array, size_t size, int target) {
    // Preconditions
    assert(array != NULL);
    assert(size > 0);
    // Assume array is sorted (can't easily assert this)
```

```
    size_t left = 0, right = size - 1;

    while (left <= right) {
        size_t mid = left + (right - left) / 2;

        // Invariant: target must be in range [left, right] if present
        assert(mid < size);

        if (array[mid] == target) {
            printf("Found target at index %zu\n", mid);
            return;
        } else if (array[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    printf("Target not found\n");
}
```

These assertions serve multiple debugging purposes. The precondition assertions catch invalid inputs immediately at function entry, preventing the function from operating on invalid data. The invariant assertion within the loop catches logic errors in the search algorithm itself. If the binary search logic were incorrect, the invariant assertion would trigger, providing immediate feedback about where the algorithm went wrong.

Design by contract extends assertions to formal specifications, creating a comprehensive framework for expressing and verifying program correctness:

- *Preconditions*: What must be true when function is called. These help debug caller errors and interface misunderstandings.
- *Postconditions*: What the function guarantees upon return. These help debug implementation errors within the function.
- *Invariants*: What remains true throughout execution. These help debug algorithmic errors and state consistency problems.

Structured Logging for Debugging

Structured logging provides consistent, searchable debugging information that aids both development and production troubleshooting. The key insight behind

structured logging is that log messages should be machine-readable as well as human-readable, enabling powerful analysis and correlation capabilities that are essential for debugging complex systems.

Traditional free-form log messages are difficult to parse and analyze programmatically. Structured logging addresses this by using consistent formats (typically JSON) that preserve the relationships between different pieces of information and enable sophisticated querying and analysis.

```
import logging
import json
from datetime import datetime

# Configure structured logging
logging.basicConfig(
    level=logging.DEBUG,
    format='%(message)s',
    handlers=[logging.StreamHandler()]
)

def structured_log(level, event, **kwargs):
    """Log structured data as JSON"""
    log_entry = {
        'timestamp': datetime.utcnow().isoformat(),
        'level': level,
        'event': event,
        **kwargs
    }
    logging.log(getattr(logging, level.upper()), json.dumps(log_entry
        ))

# Usage examples
def process_user_request(user_id, action):
    structured_log('info', 'request_started',
                user_id=user_id, action=action)

    try:
        result = perform_action(user_id, action)
        structured_log('info', 'request_completed',
                    user_id=user_id, action=action,
                    result_size=len(result))
        return result
    except ValueError as e:
        structured_log('error', 'validation_failed',
                    user_id=user_id, action=action,
                    error=str(e))
```

```
        raise
    except Exception as e:
        structured_log('error', 'unexpected_error',
                       user_id=user_id, action=action,
                       error_type=type(e).__name__,
                       error_message=str(e))
        raise
```

This structured approach enables powerful log analysis and correlation, particularly valuable in production debugging scenarios. By maintaining consistent field names and formats, structured logs can be automatically parsed, indexed, and queried. This makes it possible to trace request flows, correlate errors with system state, and identify patterns that would be invisible in traditional log formats.

3.6.2 Production and Live System Debugging

Production debugging presents unique challenges: limited access, performance constraints, and the need to maintain system availability while investigating issues. Traditional debugging techniques must be adapted for these environments where stopping execution or modifying code is often impossible.

The fundamental challenge of production debugging is observing system behaviour without disrupting normal operation. This requires a shift from intrusive debugging techniques (like breakpoints and single-stepping) to observational approaches that gather information passively and provide insights into system behaviour over time.

Observability and Monitoring

Observability encompasses the tools and practices that make system behaviour visible and understandable in production environments. The three pillars of observability—metrics, logs, and traces—work together to provide comprehensive insight into system operation. Each pillar offers different perspectives on system behaviour, and their combination enables effective debugging of complex distributed systems.

The philosophy of observability is that systems should be designed to be understood. This means building in the capability to observe system behaviour from the beginning, rather than trying to add debugging capabilities after problems arise.

Metrics provide quantitative measurements of system behaviour over time. They answer questions like "how fast is the system running?" and "how often

do errors occur?" Metrics are particularly valuable for identifying performance trends and operational patterns that might indicate developing problems before they become critical:

```
import time
from collections import defaultdict
from threading import Lock

class MetricsCollector:
    def __init__(self):
        self.counters = defaultdict(int)
        self.gauges = defaultdict(float)
        self.histograms = defaultdict(list)
        self.lock = Lock()

    def increment(self, name, value=1, tags=None):
        """Increment a counter metric"""
        with self.lock:
            key = self._make_key(name, tags)
            self.counters[key] += value

    def gauge(self, name, value, tags=None):
        """Set a gauge metric"""
        with self.lock:
            key = self._make_key(name, tags)
            self.gauges[key] = value

    def histogram(self, name, value, tags=None):
        """Record a histogram value (e.g., response time)"""
        with self.lock:
            key = self._make_key(name, tags)
            self.histograms[key].append(value)

    def _make_key(self, name, tags):
        if tags:
            tag_str = ','.join(f'{k}={v}' for k, v in sorted(tags.
                items()))
            return f'{name},{tag_str}'
        return name

# Usage in application code
metrics = MetricsCollector()

def handle_request(request_type):
    start_time = time.time()
```

```
    try:
        # Process request
        result = process_request(request_type)

        # Record success metrics
        metrics.increment('requests_total',
                          tags={'type': request_type, 'status': 'success'
                              })

    except Exception as e:
        # Record error metrics
        metrics.increment('requests_total',
                          tags={'type': request_type, 'status': 'error'
                              })
        metrics.increment('errors_total',
                          tags={'type': request_type, 'error': type(e).
                              __name__})
        raise
    finally:
        # Record timing
        duration = time.time() - start_time
        metrics.histogram('request_duration_seconds', duration,
                          tags={'type': request_type})
```

This metrics collection approach provides several debugging advantages. Counter metrics reveal the frequency of different events, helping identify unusual patterns or spikes in activity. Gauge metrics capture point-in-time values like memory usage or queue length, revealing resource constraints or capacity issues. Histogram metrics preserve the distribution of values like response times, enabling analysis of performance characteristics and identification of outliers.

Distributed tracing tracks requests across multiple services, providing visibility into the complete flow of operations in complex systems. Tracing is essential for debugging distributed systems because it connects related operations across service boundaries, making it possible to understand how a single user request flows through multiple components:

```
import uuid
from contextlib import contextmanager
from threading import local

class TraceContext:
    def __init__(self):
```

```
        self.local = local()

    def start_span(self, operation_name, trace_id=None,
         parent_span_id=None):
        """Start a new span"""
        span_id = str(uuid.uuid4())
        if trace_id is None:
            trace_id = str(uuid.uuid4())

        span = {
            'trace_id': trace_id,
            'span_id': span_id,
            'parent_span_id': parent_span_id,
            'operation_name': operation_name,
            'start_time': time.time(),
            'tags': {}
        }

        if not hasattr(self.local, 'span_stack'):
            self.local.span_stack = []

        self.local.span_stack.append(span)
        return span

    @contextmanager
    def span(self, operation_name):
        """Context manager for automatic span management"""
        current_span = self.get_current_span()
        trace_id = current_span['trace_id'] if current_span else None
        parent_span_id = current_span['span_id'] if current_span else
             None

        span = self.start_span(operation_name, trace_id,
             parent_span_id)
        try:
            yield span
        finally:
            self.finish_span(span)

    def finish_span(self, span):
        """Finish and log a span"""
        span['end_time'] = time.time()
        span['duration'] = span['end_time'] - span['start_time']

        # Log span data (in practice, send to tracing system)
```

```
        structured_log('info', 'span_completed', **span)

        if hasattr(self.local, 'span_stack'):
            self.local.span_stack.pop()

tracer = TraceContext()

def database_operation(query):
    with tracer.span('database_query') as span:
        span['tags']['query_type'] = query.split()[0].upper()
        span['tags']['table'] = extract_table_name(query)

        # Simulate database operation
        result = execute_query(query)
        span['tags']['result_count'] = len(result)
        return result
```

This tracing implementation creates a hierarchical view of operations within a request, with each span representing a discrete operation. The trace ID connects all spans that belong to the same logical request, while parent-child relationships between spans show the call hierarchy. This structure enables debugging complex interactions by providing a complete timeline of operations and their relationships.

Remote Debugging Techniques

Production systems often require debugging without direct access to the running environment. Remote debugging techniques enable investigation while minimising system disruption. The key principle is to provide controlled access to internal system state and behaviour without compromising security or stability.

Remote debugging techniques must balance the need for diagnostic information with the constraints of production environments. This typically means providing read-only access to system state, enabling configuration changes that affect logging or monitoring behaviour, and offering safe ways to trigger diagnostic operations.

Debug endpoints provide controlled access to system state through HTTP APIs that can be queried from external tools. These endpoints should be carefully designed to avoid exposing sensitive information while providing useful diagnostic data:

```
from flask import Flask, jsonify, request
import psutil
import threading
```

```
import gc

app = Flask(__name__)

@app.route('/debug/health')
def health_check():
    """Basic health information"""
    return jsonify({
        'status': 'healthy',
        'uptime': time.time() - start_time,
        'memory_usage': psutil.Process().memory_info().rss,
        'cpu_percent': psutil.Process().cpu_percent(),
        'thread_count': threading.active_count()
    })

@app.route('/debug/metrics')
def get_metrics():
    """Expose application metrics"""
    return jsonify({
        'counters': dict(metrics.counters),
        'gauges': dict(metrics.gauges),
        'histogram_summaries': {
            k: {
                'count': len(v),
                'avg': sum(v) / len(v) if v else 0,
                'min': min(v) if v else 0,
                'max': max(v) if v else 0
            } for k, v in metrics.histograms.items()
        }
    })

@app.route('/debug/gc')
def garbage_collect():
    """Force garbage collection (use carefully)"""
    before = len(gc.get_objects())
    collected = gc.collect()
    after = len(gc.get_objects())

    return jsonify({
        'objects_before': before,
        'objects_after': after,
        'objects_collected': collected
    })

# Only enable in debug mode
```

```
if app.debug:
    app.register_blueprint(debug_routes, url_prefix='/debug')
```

These debug endpoints serve different purposes in production debugging scenarios. The health endpoint provides immediate visibility into basic system vitals, helping determine if performance issues are related to resource constraints. The metrics endpoint exposes application-level measurements that reveal operational patterns and anomalies. The garbage collection endpoint provides a way to investigate memory-related issues by triggering cleanup operations and measuring their effects.

Dynamic configuration allows runtime behaviour modification without deploying new code. This capability is crucial for production debugging because it enables investigators to adjust logging levels, enable detailed monitoring, or activate alternative code paths to gather more information about problematic behaviour:

```
class ConfigManager:
    def __init__(self):
        self.config = {
            'debug_level': 'INFO',
            'enable_detailed_logging': False,
            'feature_flags': {
                'new_algorithm': False,
                'enhanced_validation': True
            }
        }
        self.lock = Lock()

    def get(self, key, default=None):
        with self.lock:
            keys = key.split('.')
            value = self.config
            for k in keys:
                value = value.get(k, default)
                if value is default:
                    break
            return value

    def set(self, key, value):
        with self.lock:
            keys = key.split('.')
            config = self.config
            for k in keys[:-1]:
```

```
                config = config.setdefault(k, {})
            config[keys[-1]] = value

    def reload_from_file(self, filename):
        """Reload configuration from external file"""
        try:
            with open(filename) as f:
                new_config = json.load(f)
            with self.lock:
                self.config.update(new_config)
            logging.info(f"Configuration reloaded from {filename}")
        except Exception as e:
            logging.error(f"Failed to reload config: {e}")

config = ConfigManager()

# Usage in application code
def process_data(data):
    if config.get('enable_detailed_logging', False):
        logging.debug(f"Processing data: {data}")

    if config.get('feature_flags.new_algorithm', False):
        return new_algorithm(data)
    else:
        return legacy_algorithm(data)
```

This configuration management approach enables several debugging strategies. Detailed logging can be enabled selectively to gather more information about specific code paths without overwhelming the system with verbose output. Feature flags allow switching between different implementations to isolate problematic behaviour. The ability to reload configuration from external files means that debugging behaviour can be adjusted without restarting services, preserving system state and ongoing investigations.

Post-Mortem Analysis

When production issues occur, systematic post-mortem analysis helps prevent recurrence and improves system resilience. The goal of post-mortem analysis is not just to fix the immediate problem, but to understand the systemic factors that allowed the problem to occur and to implement changes that prevent similar issues in the future.

Effective post-mortem analysis requires both technical investigation skills and systematic approaches to problem-solving. The technical aspects focus on

gathering and analyzing evidence about what happened, while the systematic aspects ensure that learning from the incident is captured and acted upon.

Incident response framework:

1. *Detection.* Automated monitoring alerts or user reports identify that a problem has occurred. The speed and accuracy of detection directly affects the severity of the incident.

2. *Response.* Immediate mitigation to restore service, focusing on minimizing user impact rather than understanding root causes. This phase prioritizes system stability over investigation.

3. *Investigation.* Root cause analysis using available data to understand what went wrong and why. This phase requires careful examination of logs, metrics, and system state.

4. *Resolution.* Permanent fix implementation that addresses the root cause rather than just the symptoms of the problem.

5. *Prevention.* Process improvements to prevent recurrence, including monitoring enhancements, system design changes, and operational procedure updates.

Core dump analysis for crashed applications provides detailed information about program state at the time of failure. Core dumps preserve the complete memory image of a crashed process, enabling detailed investigation of the conditions that led to the crash:

```
# Generate core dump on crash (Linux)
ulimit -c unlimited

# Compile C program with debugging information
gcc -g -o application application.c

# Run program (may crash and produce core dump)
./application

# Analyze core dump with GDB
gdb ./application core.12345
(gdb) bt              # Backtrace
(gdb) info registers  # Register state
(gdb) print variable  # Examine variables
(gdb) thread apply all bt  # All thread backtraces (if multithreaded)
```

Core dump analysis reveals information that might not be available through other means. The backtrace shows the exact sequence of function calls that led to the crash, while variable examination reveals the program state at the time of failure. For multithreaded applications, examining all thread backtraces can reveal deadlocks, race conditions, or other concurrency-related issues.

Log aggregation and analysis helps identify patterns and correlations that might not be visible when examining individual log entries. By processing large volumes of log data systematically, it becomes possible to detect error patterns, performance trends, and operational anomalies:

```
def analyze_error_patterns(log_file, time_window_minutes=10):
    """Analyze error patterns in log files"""
    errors = []

    with open(log_file) as f:
        for line in f:
            try:
                log_entry = json.loads(line)
                if log_entry.get('level') == 'ERROR':
                    errors.append(log_entry)
            except json.JSONDecodeError:
                continue

    # Group errors by time windows
    time_buckets = defaultdict(list)
    for error in errors:
        timestamp = datetime.fromisoformat(error['timestamp'])
        bucket = timestamp.replace(
            minute=timestamp.minute // time_window_minutes *
                time_window_minutes,
            second=0,
            microsecond=0
        )
        time_buckets[bucket].append(error)

    # Identify error spikes
    avg_errors = sum(len(bucket) for bucket in time_buckets.values())
         / len(time_buckets)
    spikes = {
        timestamp: errors
        for timestamp, errors in time_buckets.items()
        if len(errors) > avg_errors * 2 # 2x average threshold
    }
```

```
    return {
        'total_errors': len(errors),
        'average_per_window': avg_errors,
        'error_spikes': spikes,
        'common_errors': most_common_errors(errors)
    }
```

This analysis approach reveals temporal patterns in error occurrences that might indicate systemic issues. Error spikes might correlate with external events like traffic surges or deployment activities. Common error patterns might indicate design flaws or operational issues that need addressing. By processing log data systematically, it becomes possible to distinguish between isolated incidents and systemic problems.

Performance Debugging in Production

Performance issues in production require specialised techniques that minimise impact while gathering diagnostic information. Unlike functional bugs that cause obvious failures, performance problems often manifest as gradual degradation that can be difficult to isolate and diagnose.

The challenge of production performance debugging is gathering sufficient information to understand system behaviour without significantly impacting the performance being investigated. This requires techniques that provide statistical sampling rather than comprehensive monitoring, and that can be enabled and disabled dynamically based on investigation needs.

Sampling profilers collect performance data with minimal overhead by periodically sampling program execution rather than monitoring every operation. This approach provides statistical insight into where the program spends its time while adding minimal performance impact:

```
import signal
import sys
import traceback
from collections import defaultdict

class SamplingProfiler:
    def __init__(self, interval=0.1):
        self.interval = interval
        self.samples = defaultdict(int)
        self.enabled = False
```

```
    def _sample_handler(self, signum, frame):
        """Signal handler to collect stack samples"""
        if self.enabled:
            # Get current stack trace
            stack = traceback.extract_tb(frame.f_back)
            stack_key = '|'.join(f"{s.filename}:{s.name}:{s.lineno}"
                                 for s in stack[-5:]) # Last 5 frames
            self.samples[stack_key] += 1

    def start(self):
        """Start profiling"""
        self.enabled = True
        signal.signal(signal.SIGALRM, self._sample_handler)
        signal.setitimer(signal.ITIMER_REAL, self.interval, self.
            interval)

    def stop(self):
        """Stop profiling and return results"""
        self.enabled = False
        signal.alarm(0)

        total_samples = sum(self.samples.values())
        results = [
            {
                'stack': stack,
                'samples': count,
                'percentage': (count / total_samples) * 100
            }
            for stack, count in self.samples.most_common(10)
        ]

        return results

# Usage
profiler = SamplingProfiler()
profiler.start()

# Run application code...

results = profiler.stop()
for result in results:
    print(f"{result['percentage']:.1f}%: {result['stack']}")
```

This sampling approach provides insight into performance hotspots by statistically measuring where the program spends its execution time. The periodic

sampling introduces minimal overhead while providing enough data to identify performance bottlenecks. The stack trace information shows not just which functions are consuming time, but also the call contexts in which they are executed, helping understand the conditions that lead to performance problems.

This advanced debugging approach provides the tools necessary for maintaining and troubleshooting complex software systems in production environments, complementing the foundational debugging techniques covered earlier in earlier sections.

3.6.3 Summary

Advanced debugging practices extend beyond development-time techniques to encompass error prevention and production system maintenance. Error handling strategies, including defensive programming, structured exception handling, and comprehensive logging, serve as both debugging prevention mechanisms and diagnostic aids when properly implemented with clear understanding of their purpose and application. Production debugging requires specialised approaches such as observability systems that make system behaviour visible, remote debugging capabilities that provide safe access to system internals, and systematic post-mortem analysis frameworks that transform incidents into learning opportunities and systemic improvements. These techniques are essential for building resilient software systems that can be effectively maintained and debugged throughout their operational lifecycle, ensuring that problems can be identified, understood, and resolved efficiently while maintaining system reliability and performance.

3.7 The Impact of LLMs

The rise of large language models (LLMs)—AI systems trained on vast code repositories and capable of natural language reasoning—is transforming software engineering practices. Tools like GitHub Copilot, Cursor, or integrated IDE plugins (e.g., in VS Code or JetBrains) leverage LLMs to assist in real-time, shifting debugging, optimisation, and testing from purely manual processes to AI-augmented workflows. While traditional methods remain foundational, LLMs introduce efficiency gains, new risks, and a reevaluation of developer skills.

3.7.1 Debugging

Traditional debugging relies on tools like print statements, breakpoints, and core dumps to isolate issues (as discussed in Section 3.3.3 and Section 3.6.2). LLMs enhance this by analysing code, error logs, or stack traces to suggest hypotheses and fixes conversationally.

- *Automated Hypothesis Generation.* Describe a bug in natural language (e.g., "My loop crashes on large inputs"), and an LLM can simulate execution paths, identify potential causes (e.g., overflow or off-by-one errors), and propose patches. This accelerates the "develop a hypothesis" stage (Section 3.3.1).
- *Conversational Workflows.* Integrated tools allow "chatting" with the code, e.g., querying variable states or explaining runtime behaviour without manual stepping.
- *Consequences.* Debugging becomes faster and more accessible for novices, reducing time from hours to minutes. However, over-reliance risks "hallucinated" suggestions (inaccurate fixes) or eroded low-level skills. In regulated fields like healthcare, human validation remains essential to avoid subtle errors.

Example: Using an LLM in a Python debugger session:

```
# User prompt to LLM: "Debug why this divides by zero"
def divide(x, y):
    return x / y

# LLM response: "Potential runtime error if y=0.
# Suggest adding: if y == 0: raise ValueError('Division by zero')"
```

3.7.2 Optimisation

Optimisation traditionally involves profiling (e.g., gprof in Section 3.4) and techniques like loop unrolling or memory pooling (Sections 3.4.1 and 3.4.2). LLMs automate pattern recognition from codebases, suggesting refactors or even generating optimised variants.

- *Proactive Suggestions:* LLMs can analyse bottlenecks (e.g., via integrated profilers) and recommend algorithmic improvements, such as replacing $O(n^2)$ loops with $O(n \log n)$ alternatives, or hardware-aware tweaks like GPU offloading.
- *Iterative Auto-Optimisation:* Future tools may create "optimisation loops," where LLMs test code variants in simulated environments, evolving solutions via techniques like genetic algorithms.
- *Consequences:* This democratises optimisation, enabling non-experts to achieve efficiency gains (e.g., 20–40% faster code in benchmarks). Trade-offs include increased code complexity if suggestions aren't reviewed, or biases

from training data favouring certain languages/architectures. For dynamic programming (Section 3.4.3), LLMs excel at generating memoization or tabulation code but may overlook domain-specific constraints.

In machine learning contexts (e.g., confusion matrices in Section 3.4.4), LLMs can debug models by interpreting metrics and suggesting hyperparameter tweaks, blending optimisation with ML-specific evaluation.

3.7.3 Testing

Testing emphasises coverage and types like unit or regression tests (Section 3.5). LLMs generate test cases automatically, inferring edge cases from code or specs.

- *Automated Test Generation:* Given a function, an LLM can produce unit tests, property-based tests (e.g., via Hypothesis), or fuzz inputs, boosting coverage without manual effort.
- *Self-Testing and CI Integration:* LLMs can prioritise tests in pipelines (Section 3.5.1) or even write "self-healing" tests that adapt to code changes.
- *Consequences:* Testing cycles shorten dramatically, with studies showing 30–50% higher coverage. Risks include incomplete tests missing nuanced behaviours or biases amplifying data imbalances (e.g., in confusion matrices). For property-based and fuzz testing, LLMs enhance randomness but require validation to avoid false positives.

Example: LLM-generated test for a Fibonacci function (from Section 3.4.3):

```
# LLM prompt: "Generate unit tests for fib(n)"
def test_fib():
    assert fib(0) == 0
    assert fib(1) == 1
    assert fib(5) == 5 # Plus edge cases like negative inputs
```

3.7.4 Broader Perspectives and Challenges

LLMs shift the paradigm from tool mastery to "AI literacy"—prompt engineering, verifying outputs, and understanding limitations. Consequences include:

- *Accelerated Development:* Chapters like this may evolve to focus on hybrid workflows, with less emphasis on manual tools.
- *Ethical and Practical Risks:* Dependency on proprietary models raises privacy concerns (e.g., sharing code), while hallucinations demand rigorous verification.

- *Future Evolution:* By 2025+, multimodal LLMs (handling code + visuals) could enable "visual debugging" or automated hardware optimisation (e.g., for Raspberry Pi Pico in Section 3.3.3).
- *Skill Shifts:* Developers must balance AI assistance with fundamentals to avoid skill atrophy, much like how calculators changed math education.

While LLMs don't replace core concepts, they augment them, promising more robust software but requiring vigilance. Experiment with tools like Claude, Grok or Copilot to see these changes firsthand.

3.7.5 Summary

Large language models (LLMs) are revolutionising debugging, optimisation, and testing in software development. In debugging, LLMs automate hypothesis generation and suggest fixes by analysing code and errors, speeding up the process but risking over-reliance and inaccurate suggestions. For optimisation, they propose efficient algorithms and hardware-aware tweaks, potentially automating iterative refinement, though they may introduce complexity or bias. In testing, LLMs generate comprehensive test cases, including unit and fuzz tests, boosting coverage but requiring validation to avoid incomplete or biased tests. Consequences include faster development, democratised access for novices, and new risks like dependency on proprietary models, privacy concerns, and skill erosion. The chapter's core concepts remain vital, but LLMs shift workflows toward AI-augmented collaboration, necessitating "AI literacy" alongside traditional skills.

3.8 Practice

Workbook: Chapter 3.
https://github.com/Feyerabend/bb/tree/main/workbook/ch03
Scan the QR code to access exercises, code, resources and updates.

Exercises

1. *Explain the limitations of debugging using print statements and compare it with the advantages of using breakpoints in a debugging tool like GDB.*

   Understanding the pros and cons of different debugging techniques can help you decide the best tool for various scenarios.

2. *Describe a situation where optimising for memory usage could negatively impact speed, and vice versa. Provide an example from your own experience or research.*

   Knowing the trade-offs between memory and speed is crucial in writing efficient code.

3. *Implement an algorithm that optimises memory usage by using a more efficient data structure, and explain how it improves performance.*

   Understanding how data structure selection impacts memory efficiency is key to writing optimised code.

4. *Explore limitations of verification and testing by examining cases where testing alone cannot ensure program correctness.*

   Understanding these boundaries emphasises the need for complementary methods, such as formal verification, in developing robust software systems. Also explore for a deeper understanding what verification entails.

5. *Explore other debugging tools for machine learning, and demonstrate how they aid in model evaluation and improvement.*

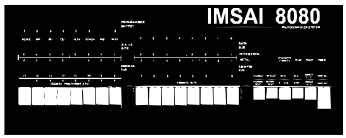
   Using tools like feature importance, residual analysis, or partial dependence plots allows for deeper insights into model behaviour and error sources. This approach can reveal underlying data issues, feature interactions, and misclassification patterns that may be missed by traditional debugging, ultimately guiding more targeted model refinements and improving overall performance.

6. *Explore additional types of end-user tests.*

   Investigate their purposes and methods, including those not covered in this chapter, such as End-to-End (E2E), usability, and acceptance tests. Mastery in these areas is essential for advancing in software development.

Example: AI and Tic-Tac-Toe

In the 1983 movie *WarGames*, David Lightman uses his IMSAI 8080 to hack a military AI, WOPR, mistaking it for a game system. To prevent nuclear war, he has the computer play tic-tac-toe against itself. The AI learns that some games, like war, are unwinnable, concluding: "A strange game. The only winning move is not to play."



Few things happen in isolation. Technology and society are in a constant feedback loop, each shaping the other. The military, once a major driver of technological progress, has long been shaped by computing innovations, which have in turn redefined warfare. As artificial intelligence evolves, we can expect dramatic shifts in how wars are fought—through autonomous systems and AI-driven strategies. Yet, despite these rapid advancements, some things remain unchanged. The process of debugging, for example, still echoes the old IMSAI 8080 panel—single-step, halt, examine, reset—reminding us that, even in an age of cutting-edge technology, foundational problem-solving techniques endure. Could this be a matter of time?

Projects: Build your own toolkit

Why build your own tools when you could rely on powerful LLMs? It's a fair question, given the transformative potential of integrating LLMs into development environments. Such models can assist with debugging, optimisation, and testing by offering real-time insights, code suggestions, and even proactive error prevention. APIs already allow developers to leverage these capabilities, but they come with trade-offs: financial costs, potential dependency on proprietary systems, and occasional inaccuracies or gaps in contextual understanding. Blind reliance on AI tools can also risk superficial problem-solving, detaching us from the core craft of programming.

Meanwhile, the act of creating our own tools offers distinct advantages. Like a craftsperson perfecting their work with specialised jigs, developers use purpose-built tools to manage complexity, ensure reliability, and optimise performance. These tools—whether they are logging, debuggers, profilers, testing frameworks, or something else—aren't just utilities; they embody a deeper understanding of our processes and goals. Building our own solutions empowers us to refine our skills, preserve control, and create systems tailored to our needs. By balancing the strengths of AI with the discipline of craftsmanship, we ensure our development processes remain precise, resilient, but also our own.

To build custom debugging tools, developers should focus on creating solutions tailored to their specific needs, especially when existing tools don't meet the unique requirements of their project. Custom tools can provide better control, efficiency, and automation, especially when dealing with complex systems, repetitive tasks, or domain-specific challenges.

When to build your own tools

- Existing tools don't fit your needs (e.g., specialised debugging for performance bottlenecks, distributed systems, or custom protocols).

- Repetitive debugging tasks slow you down (e.g., parsing logs or checking specific conditions).
- Collaboration and workflow in large teams require tools that integrate with internal processes.

How to build custom debugging tools

- Understand the problem: Identify the specific issue you want to address (e.g., memory leak, slow performance, or error tracking).
- Choose the right approach: Instrumentation, log parsing, visual dashboards, or automated monitoring can be useful depending on your need.
- Leverage existing tools: Use libraries, frameworks, and scripts to help you get started (e.g., logging libraries, profilers, or error tracking services).
- Automate repetitive tasks: Write scripts or tools that reduce manual intervention (e.g., log parsing, real-time monitoring).
- Test and iterate: Continuously refine your tool to ensure reliability and robustness in different debugging scenarios.