COMP2221 — Sys. Prog.

Polygona Graphics Engine: Pointers and Performance

Practical 04 • 03 November 2025

Instructor: Dr. Stuart James (stuart.a.james@durham.ac.uk)

Scenario

You are working on *Polygona*, a lightweight rendering engine that generates thousands of shapes and colours in real time. Every frame, large arrays of vertex data are created, updated, and passed through memory. To keep animations smooth, the system must handle these data structures with precision and efficiency — each address calculation, pointer movement, and memory allocation contributes to overall performance. Understanding how data flows through memory at this level is essential for ensuring that Polygona runs reliably, even under heavy computational load.

Learning Objectives

- Construct, trace, and reason about programs that use pointers and pointer arithmetic.
- Explain how arrays, pointers, and double pointers differ in behaviour and purpose.
- Investigate the effect of memory-access patterns on program performance.
- Design and document a reproducible benchmark using external timing tools.

Prerequisites

- Access to Linux, macOS, or `mira.dur.ac.uk`.
- You are comfortable working in a UNIX shell and editing source files.
- You have completed earlier practicals on Makefiles, basic I/O, and `struct` usage.

Tasks

Task 1: Pointer arithmetic explorer

SOLO INTRO 20m

Outcome: Write a small C program that demonstrates pointer arithmetic and explains how pointer increments relate to element size.

▶ **Activity**

Write a short program that:

- creates an array of five integers with easily recognisable values;
- uses a pointer to traverse this array, printing each element and its address;
- demonstrates how the pointer changes when incremented.

Include comments explaining why each address differs by a specific number of bytes. Verify your observations experimentally.

Task 2: Vector multiplication

SOLO INTERMEDIATE 20m

Outcome: Implement element-wise multiplication of two integer arrays using pointer arithmetic.

▶ **Activity**

Write a program that:

- defines two equal-length integer arrays and one result array,
- multiplies corresponding elements using pointer operations only,
- prints the resulting array.

Avoid array indexing syntax (`[i]`). Comment on how pointer traversal differs conceptually from array indexing. Test your program for correctness using different array sizes.

Task 3: Double pointers

SOLO INTERMEDIATE 50m

Outcome: Show how a function can allocate memory for a variable defined in another scope.

► Activity

In this task you will design a function that can **allocate memory for a variable defined in another function**. To achieve this, you will need to work with a **pointer to a pointer**—a variable that stores the address of another pointer. This concept, called *double indirection*, is used widely in systems code to build flexible memory managers, linked lists, and dynamically sized buffers.
Your goal is to write a small program where:

- a function dynamically allocates space for an integer on the heap,
- the calling function can access that memory and read the stored value,
- the program eventually releases the memory correctly.

Before writing any code:

1. Sketch a diagram showing how a variable on the **stack** (in `main`) could store a pointer that refers to memory on the **heap**.
2. Identify which part of your program will hold the pointer itself, and which part will hold the allocated data.

When implementing:

- Decide what type the parameter in your allocation function should have, and explain why a single pointer would not be sufficient.
- Include comments that clarify which variables live on the stack and which live on the heap.
- Test your program by printing the allocated value from the caller's function.
- Add a final line that releases the heap memory, explaining in a comment why this is essential.

Task 4: Assessed: Comparing pointer and array performance

ASSESSED HARD 45m

Outcome: Measure, compare, and explain the performance of pointer-based and array-based traversal across different data sizes.

► Activity

This assessment extends your work from **Task 2 (Vector multiplication)**. You will compare two versions of your program—one that uses pointer arithmetic and one that uses array indexing—to investigate how memory-access strategies affect performance as the workload increases.

Your task:

1. Use your pointer-based program from Task 2 as a baseline. Create a second version that performs the same computation using standard array indexing (`[i]`). Both programs should operate on large integer arrays (for example, 10^5 to 10^7 elements).
2. Write a short shell script called `benchmark.sh` that:
   - runs each program several times for a range of input sizes (e.g. 100 000, 1 000 000, 10 000 000);
   - uses the external `time` command to record the real execution time for each run;

     e.g. `time ./vector_pointer 1000000`

     only `real` time is needed for your analysis.
   - outputs results in a structured, Markdown-friendly format.
3. Summarise your findings in a Markdown file called `results.md`. Include:
   - a timing table comparing pointer and array versions across multiple input sizes;
   - 100 words of analysis explaining:
     - which version ran faster and how the difference scales with input size,
     - possible causes such as caching, compiler optimisation, or indexing overhead,
     - what this reveals about memory traversal in C.
4. Ensure your timing data are reproducible (run each configuration at least three times) and that your Markdown table is clearly formatted.

Assessment criteria:

- both programs produce correct results and compile successfully;
- the benchmarking script runs consistently over varying input sizes;
- results are presented clearly and interpreted with insight into system behaviour.

Deliverables:

- `vector_pointer.c` and `vector_array.c`;
- `benchmark.sh`;
- `results.md` containing timing table and written analysis.

Submission Checklist

Submission Checklist

- ☐ `pointer_arithmetic.c` – pointer arithmetic exploration (Task 1).
- ☐ `vector_multiply.c` – pointer-based vector multiplication (Task 2).
- ☐ `double_pointer.c` – function allocation using double indirection (Task 3).
- ☐ (Assessed) `vector_array.c` and `vector_pointer.c`, `benchmark.sh`, and `results.md` (Task 4).

Place all the above files in a single folder named `Practical4` and submit it to Gradescope Autograder.

```
Practical1/
|-- [Contents of Practical 1]
Practical2/
|-- [Contents of Practical 2]
Practical3/
|-- [Contents of Practical 3]
Practical4/
|-- pointer_arithmetic.c
|-- vector_multiply.c
|-- double_pointer.c
|-- vector_array.c
|-- vector_pointer.c
|-- benchmark.sh
|-- results.md
Practical5/
Practical6/
Practical7/
Practical8/
Practical9/
README.md
```

Tip

Ensure your filenames and directory structure exactly match this format so that automated scripts can locate and test your submissions. Keep each practical in its own folder and maintain your `README.md` as a record of your weekly progress.