

# Lecture 11: Performance on Performance

Bart Iver van Blokland

# From last week

- What does a `std::unique_ptr` and `std::shared_ptr` do?
- What mechanism do they use for their purpose?
- What is a memory leak?
- What is a dangling pointer?

# PSA: Guest lecture tomorrow!

- Talk about the programming language Rust
- Has some really cool ideas about memory management
  - If your program has memory issues, **it will not compile!**
- Has become the dominant modern alternative for C/C++

# Objective:

We have a program that runs slowly  
and would like to make it run faster.

The most important thing to remember from today..

# Today - performance

- **How to measure performance**

# Measuring performance

- Use the Stopwatch class you'll find in the handout code. It uses `std::chrono` to measure time.
  - If we do time measurement on the exam, you will get this class as part of the handout code.
  - `stop()` returns the number of seconds since `start()` was last called

```
class Stopwatch {  
  
    public:  
        void start();  
        double stop();  
};
```

# Measuring performance

- Let's look at some measurements:

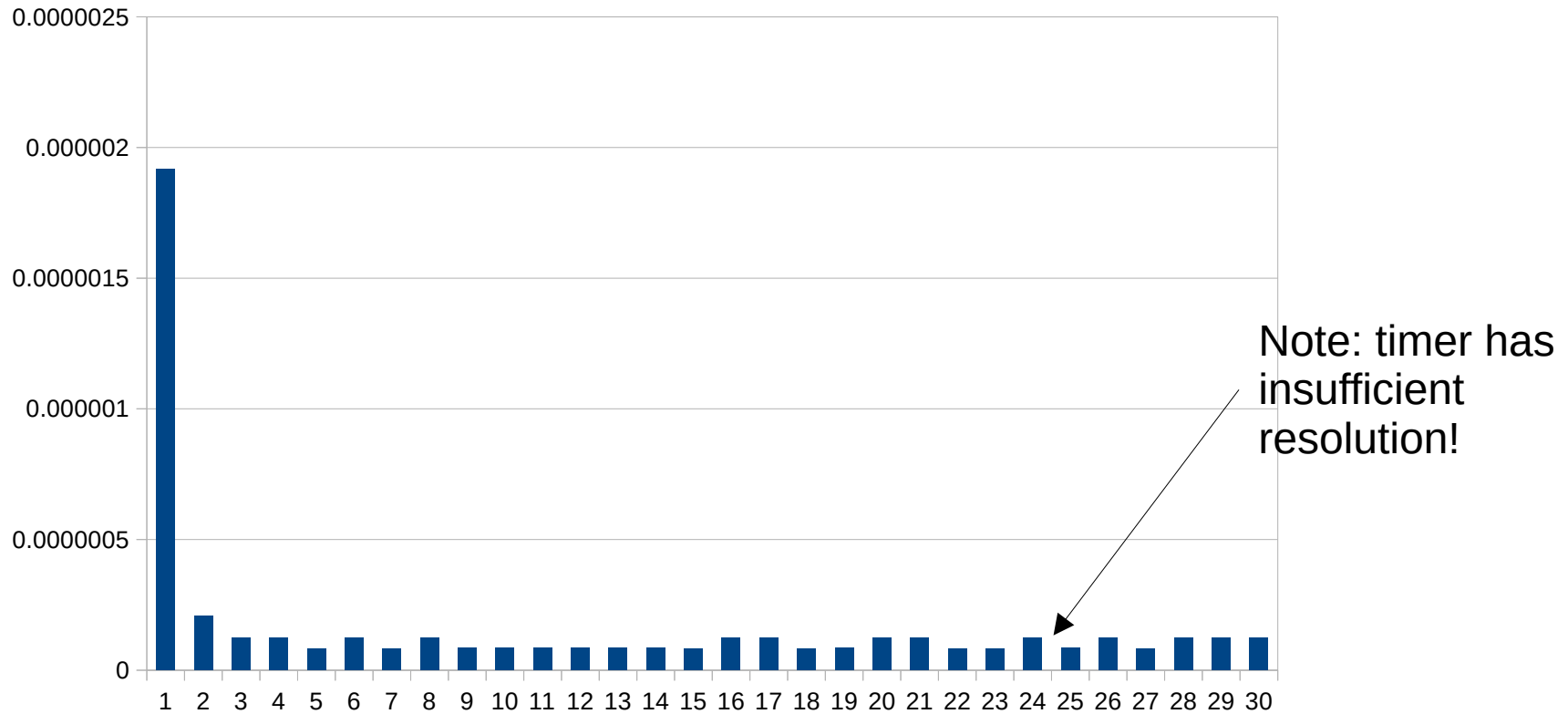
```
Stopwatch watch;  
for(int i = 0; i < 30; i++) {  
    watch.start();  
    for(int j = 0; j < 1; j++) {  
        std::string a = "hello ";  
        std::string b = "there";  
        std::string c = a + b;  
    }  
    std::cout << watch.stop() << std::endl;  
}
```

← We are measuring the execution time of these lines



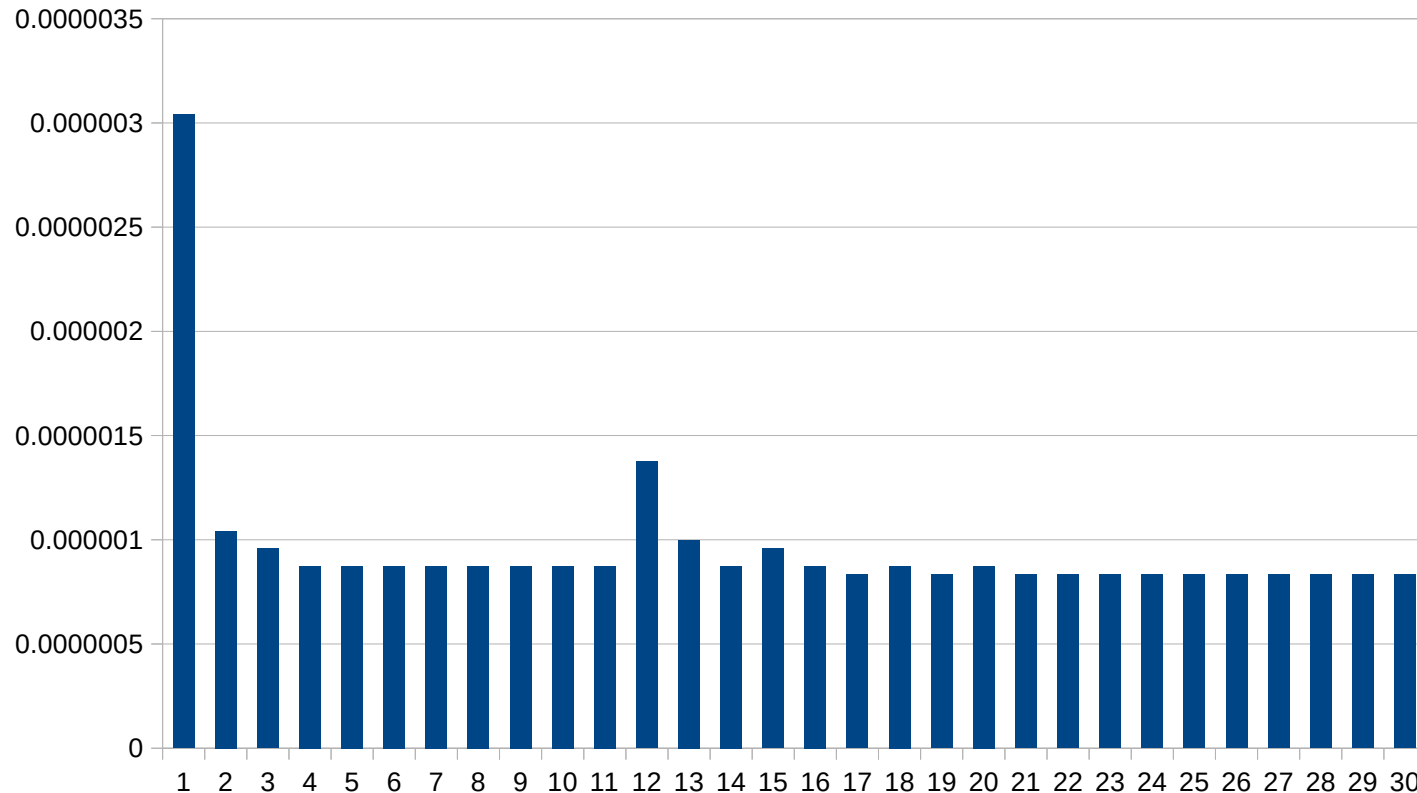
# Measuring performance

- Let's look at some measurements (1 iteration for the j loop):



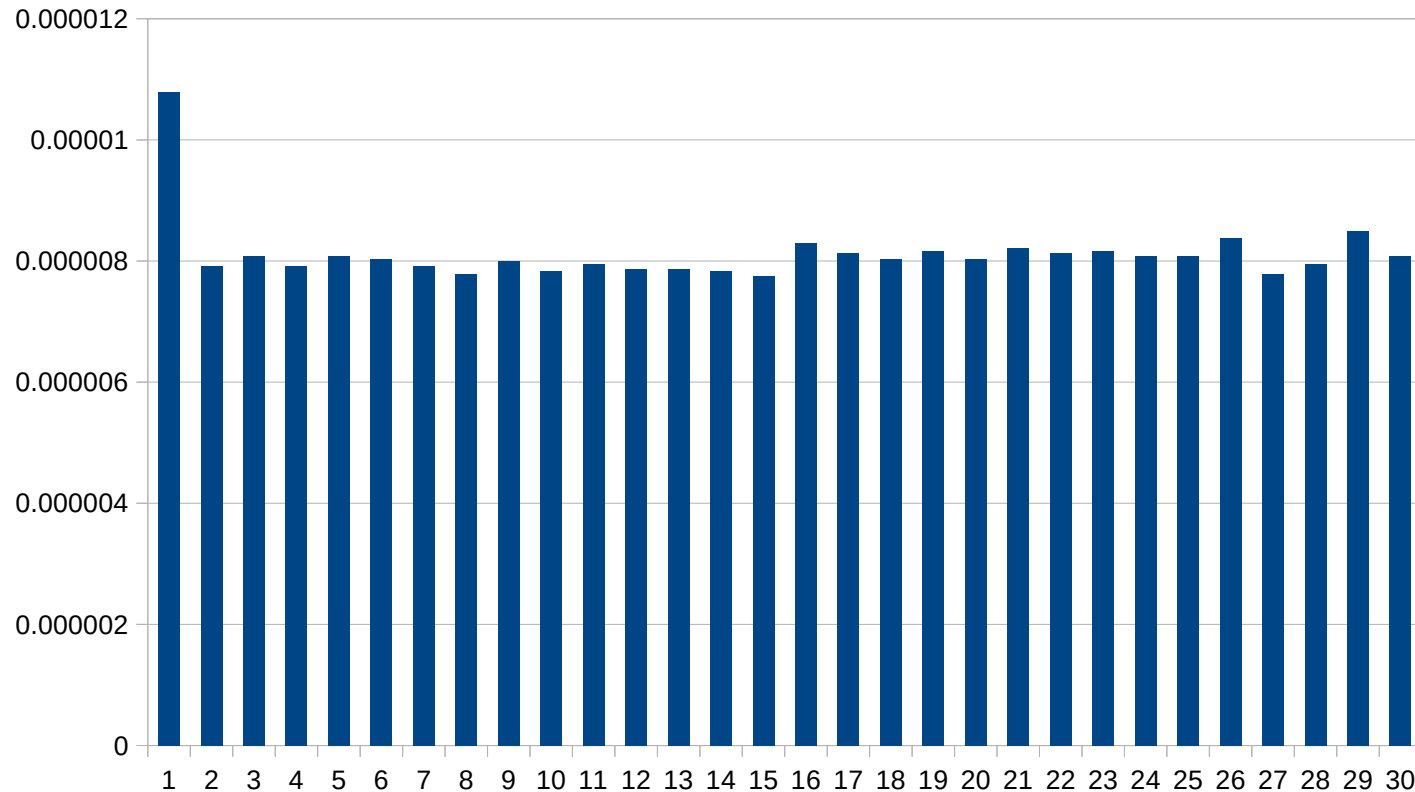
# Measuring performance

- Let's look at some measurements (10 iterations for the j loop):



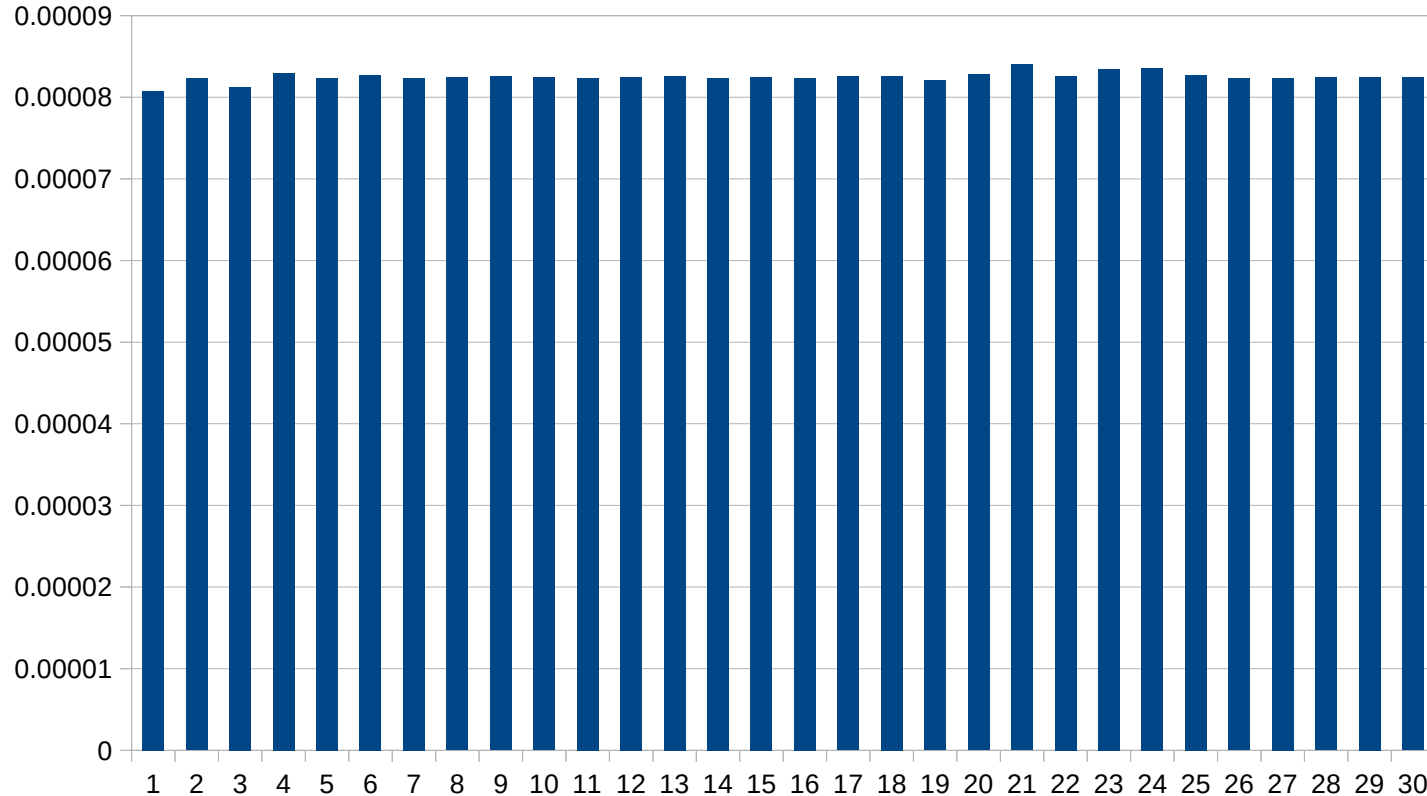
# Measuring performance

- Let's look at some measurements (100 iterations for the j loop):



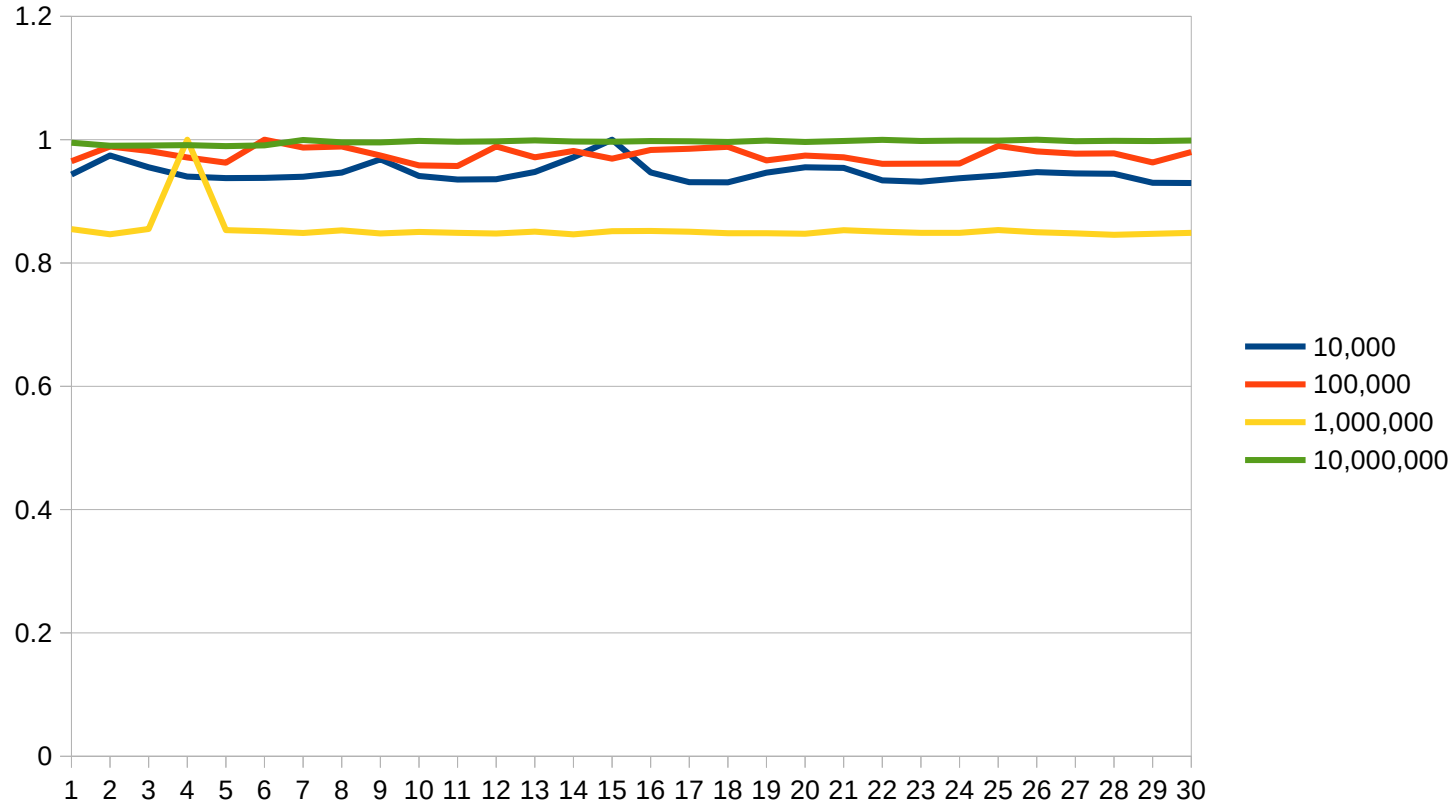
# Measuring performance

- Let's look at some measurements (1000 iterations for the j loop):



# Measuring performance

- Let's look at some measurements (> 1000 iterations for the j loop):



# Measuring performance

- Many operations take less than a nanosecond to complete. Clocks in most computers are not accurate enough to measure that time.

# Measuring performance

- Many operations take less than a nanosecond to complete. Clocks in most computers are not accurate enough to measure that time.
- Examples of factors that add noise into time measurements (there are MANY more!):
  - The operating system
  - The processor dynamically speeding up and slowing down depending on its temperature and available power
  - The type of processing core your program gets allocated to (modern CPUs can have multiple)
  - Hardware features of CPU processing cores
  - Network delays or disk access times

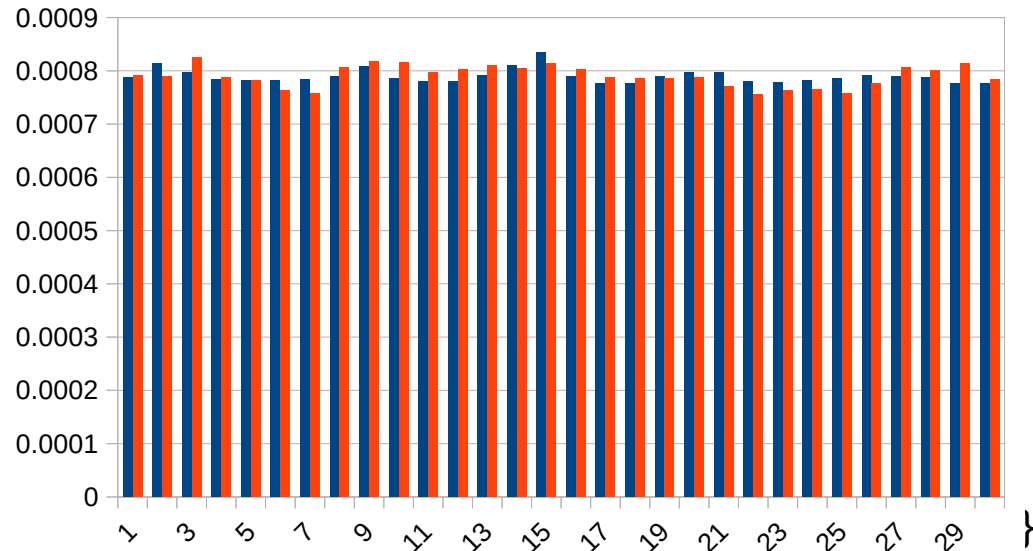
# Errors in measurements

- Random error: noise in measured results
  - Reduced by taking more samples (as we've seen)
- Systematic error: a consistent difference between the measured and true value of the metric



# Errors in measurements

The overhead of a for loop is not zero, but if what you measure takes enough time it doesn't really matter.

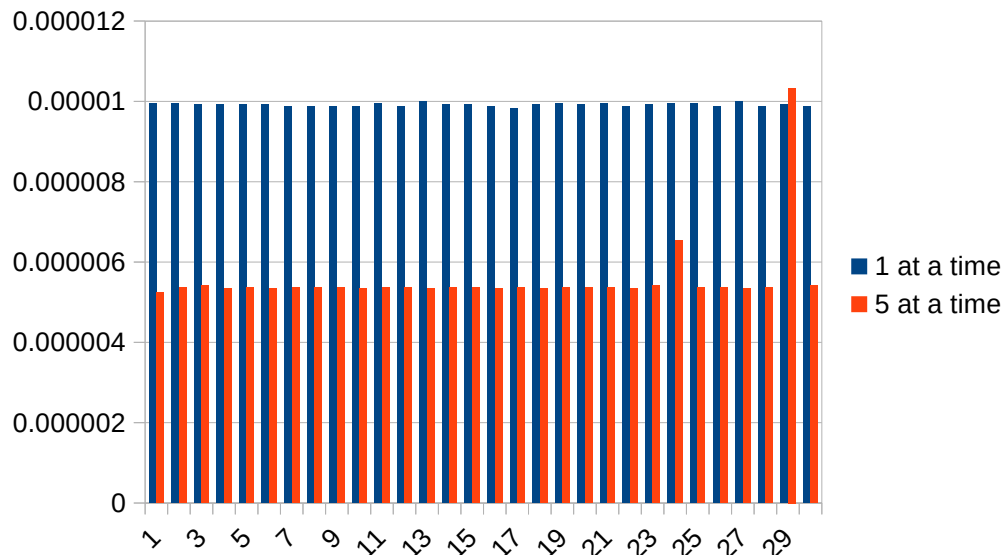


```
for(int j = 0; j < (10000 / 5); j++) {  
    std::string a = "hello ";  
    std::string b = "there";  
    std::string c = a + b;  
  
    std::string d = "hello ";  
    std::string e = "there";  
    std::string f = d + e;  
  
    std::string g = "hello ";  
    std::string h = "there";  
    std::string k = g + h;  
  
    std::string l = "hello ";  
    std::string m = "there";  
    std::string n = l + m;  
  
    std::string o = "hello ";  
    std::string p = "there";  
    std::string q = o + p;  
}
```

# Errors in measurements

Only in very special cases where the function is extremely short can it make a difference.

-> Takeaway: don't profile tiny functions!



```
for(int j = 0; j < (10000 / 5); j++) {  
    int a = 5;  
    int b = 7;  
    int c = a + b;  
  
    int d = 5;  
    int e = 7;  
    int f = d + e;  
  
    int g = 5;  
    int h = 7;  
    int k = g + h;  
  
    int l = 5;  
    int m = 7;  
    int n = l + m;  
  
    int o = 5;  
    int p = 7;  
    int q = o + p;  
}
```

# Errors in measurements

- Random error: noise in measured results
  - Reduced by taking more samples (as we've seen)
- Systematic error: a consistent difference between the measured and true value of the metric
  - Can occur in specific cases that are predominantly outside the scope of this course.

# Today

- How to measure performance
- **Tips for improving performance**

# Performance

- Can use several courses on everything there is to say on the topic
- Approach: explain steps that will get you most of the way there in most cases

# Which is faster?

```
const int matrixSize = 25000;  
std::vector<std::array<int, matrixSize>> matrix(matrixSize);
```

A

```
for(int row = 0; row < matrixSize; row++) {  
    for(int col = 0; col < matrixSize; col++) {  
        matrix.at(row).at(col) += 5;  
    }  
}
```

B

```
for(int col = 0; col < matrixSize; col++) {  
    for(int row = 0; row < matrixSize; row++) {  
        matrix.at(row).at(col) += 5;  
    }  
}
```

# Which is faster?

```
const int matrixSize = 25000;  
std::vector<std::array<int, matrixSize>> matrix(matrixSize);
```

A

```
for(int row = 0; row < matrixSize; row++) {  
    for(int col = 0; col < matrixSize; col++) {  
        matrix.at(row).at(col) += 5;  
    }  
}
```

**3.6 seconds**

B

```
for(int col = 0; col < matrixSize; col++) {  
    for(int row = 0; row < matrixSize; row++) {  
        matrix.at(row).at(col) += 5;  
    }  
}
```

**6.9 seconds**

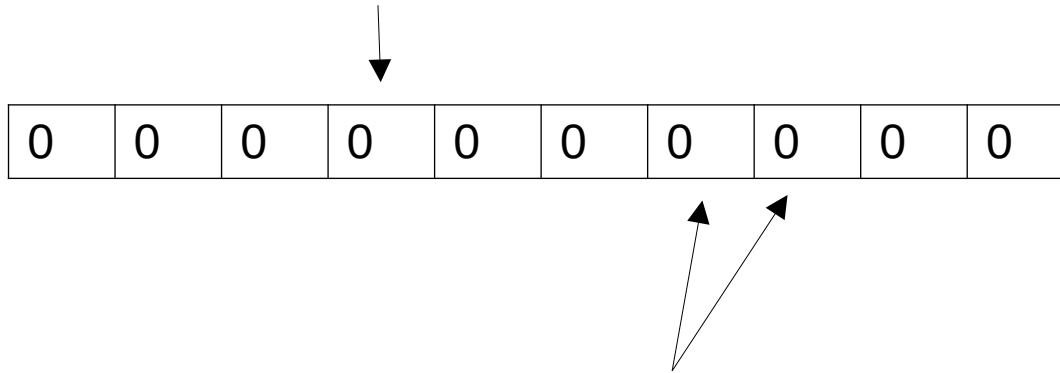
# Performance

- How to measure performance
- **Tip 1: Use the cache**



# Reminder: array layout

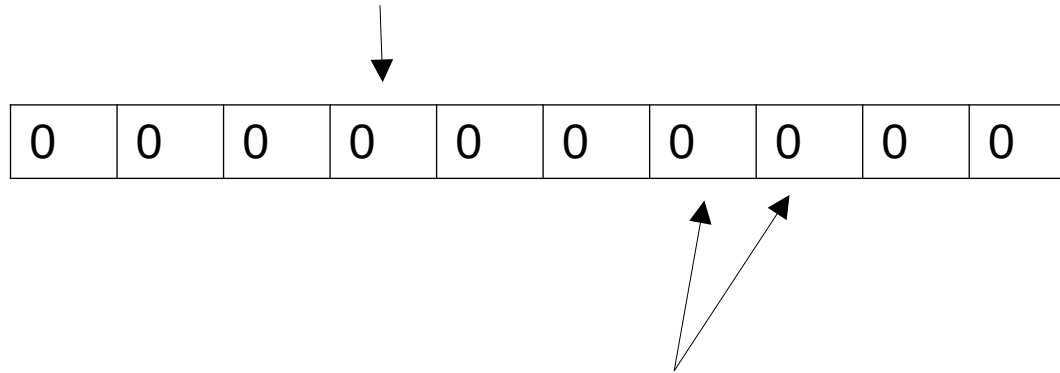
```
std::vector<int> list(10);
```



Question: given these two elements,  
what do we know about their location  
in memory?

# Reminder: array layout

```
std::vector<int> list(10);
```

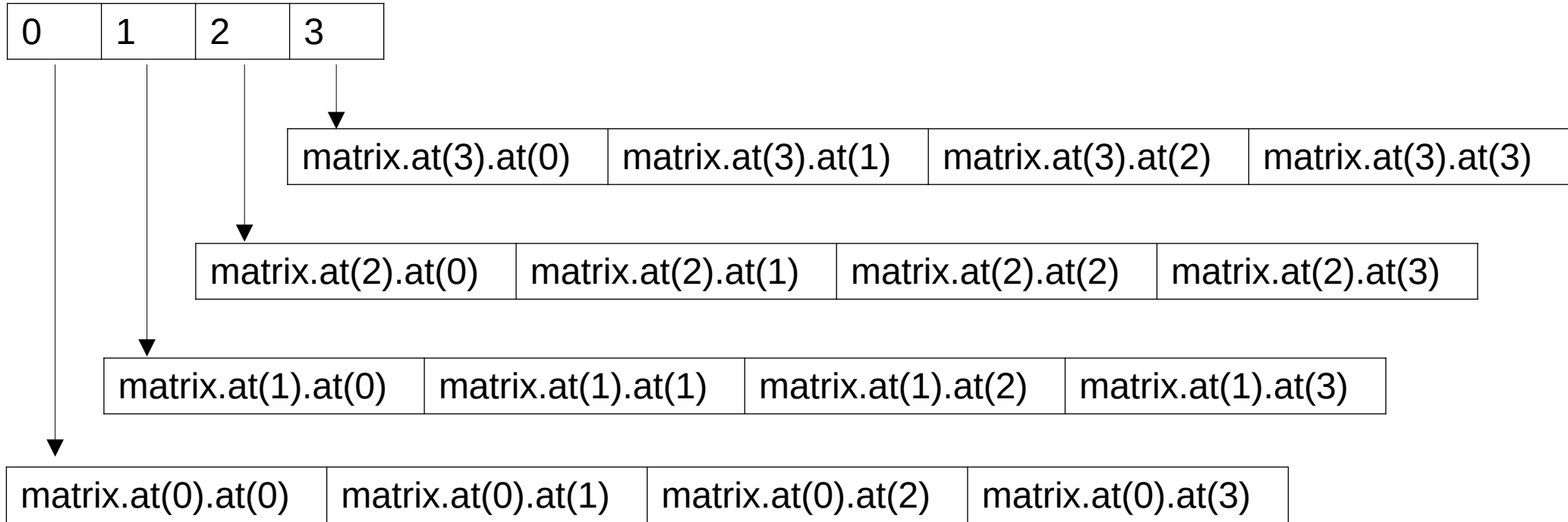


Question: given these two elements, what do we know about their location in memory?

Answer: neighbouring array elements are guaranteed to be located next to each other in memory

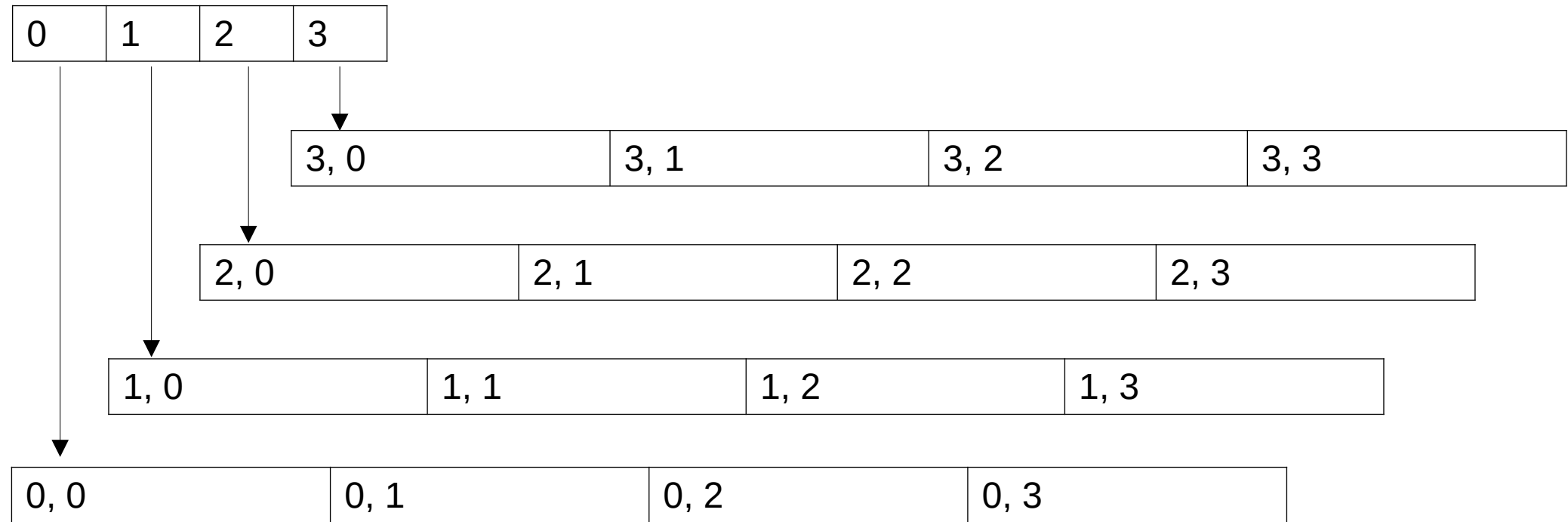
# Reminder: array layout

```
std::vector<std::array<int, 4>> matrix(4);
```



# Reminder: array layout

```
std::vector<std::array<int, 4>> matrix(4);
```

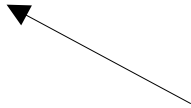


# Reminder: array layout

```
std::vector<std::array<int, 4>> matrix(4);
```

This also holds recursively. The contents of structs/classes\* or arrays are laid out next to each other in memory too.

0, 0	0, 1	0, 2	0, 3	1, 0	1, 1	1, 2	1, 3	2, 0	2, 1	2, 2	2, 3	3, 0	3, 1	3, 2	3, 3
------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------



These first 4 elements  
belong to matrix.at(0)

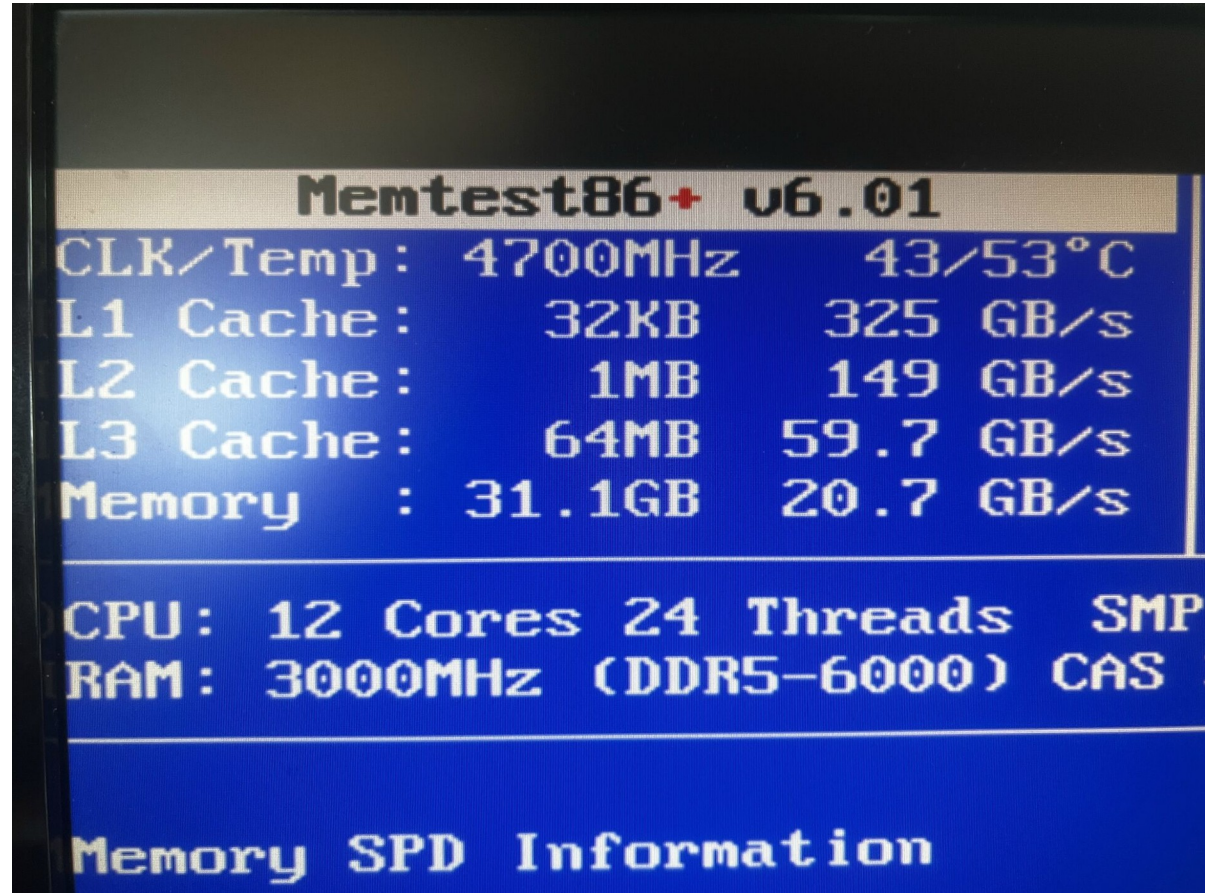
\* The standard makes no guarantees of what the layout of structs/classes are in memory, but in practice they basically always are laid out just like an array.

# The CPU cache

- The slowdown we saw is caused by two factors related to the caching of data done within the CPU.
- Question: what is the CPU cache?

# Cache

- Extremely fast memory within the processor
- Compared to the processor, RAM is slow. That is the main motivation caches exist
- Contains copies of data in RAM
- TINY in size

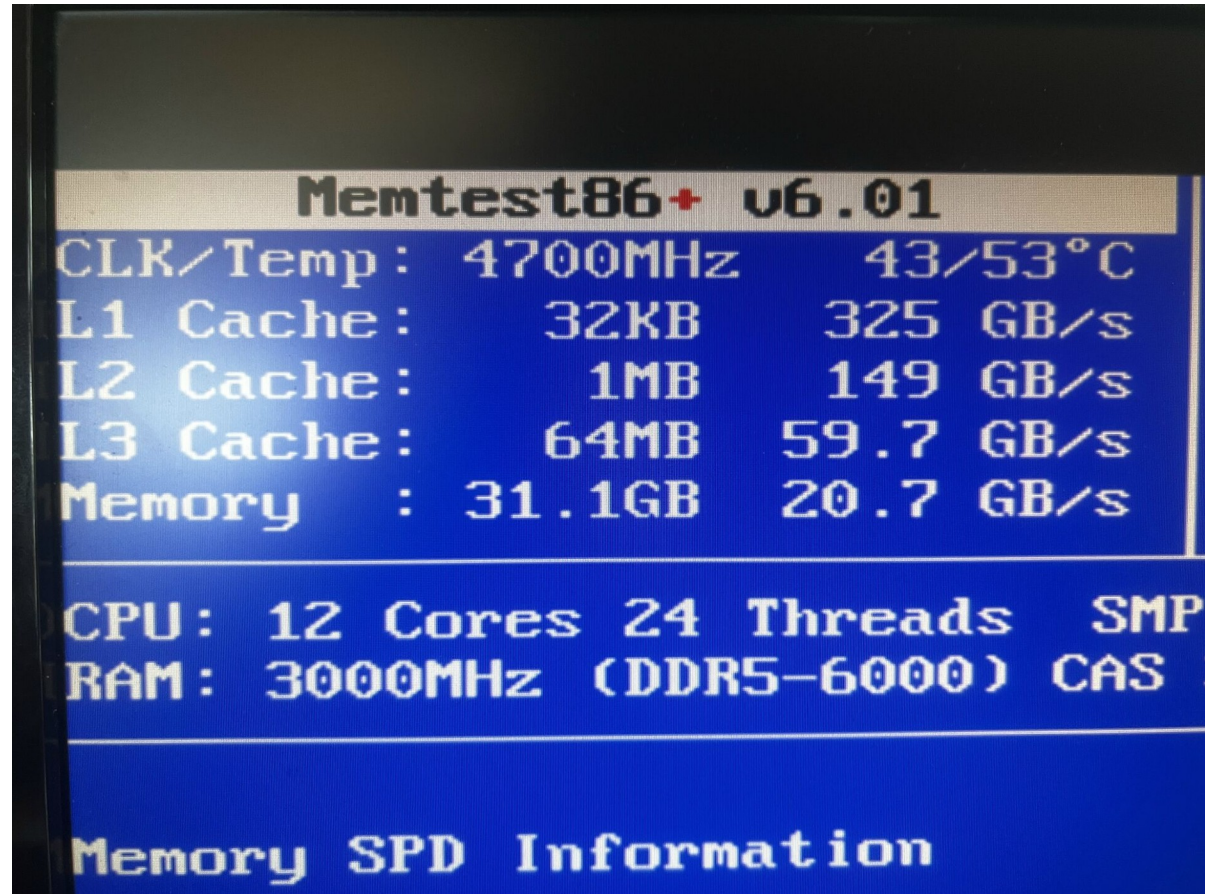


The image shows a screenshot of the Memtest86+ v6.01 boot screen. The screen has a blue background with white text. The title 'Memtest86+ v6.01' is at the top. Below it, system specifications are listed: CLK/Temp: 4700MHz 43/53°C, L1 Cache: 32KB 325 GB/s, L2 Cache: 1MB 149 GB/s, L3 Cache: 64MB 59.7 GB/s, and Memory: 31.1GB 20.7 GB/s. Further down, it shows CPU: 12 Cores 24 Threads SMP and RAM: 3000MHz (DDR5-6000) CAS. At the bottom, it says 'Memory SPD Information'.

Memtest86+ v6.01			
CLK/Temp:	4700MHz	43/53°C	
L1 Cache:	32KB	325	GB/s
L2 Cache:	1MB	149	GB/s
L3 Cache:	64MB	59.7	GB/s
Memory	: 31.1GB	20.7	GB/s
CPU: 12 Cores 24 Threads SMP			
RAM: 3000MHz (DDR5-6000) CAS			
Memory SPD Information			

# Cache: levels

- Making a cache larger also makes it slower
- Therefore, modern processors have typically 3 «levels», each larger and slower than the levels below



The image shows a screenshot of the Memtest86+ v6.01 benchmark results. The title bar is yellow with black text. The main content is on a blue background with white text. It lists various system metrics including clock speed, temperature, and cache/memory performance. Below the main table, it shows CPU and RAM specifications, and a section for memory SPD information.

Memtest86+ v6.01			
CLK/Temp:	4700MHz	43/53°C	
L1 Cache:	32KB	325	GB/s
L2 Cache:	1MB	149	GB/s
L3 Cache:	64MB	59.7	GB/s
Memory :	31.1GB	20.7	GB/s

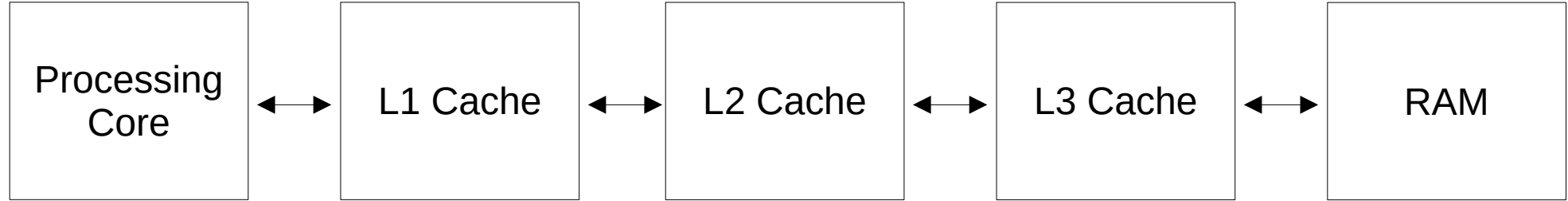
CPU:	12 Cores 24 Threads	SMP
RAM:	3000MHz (DDR5-6000)	CAS

Memory SPD Information



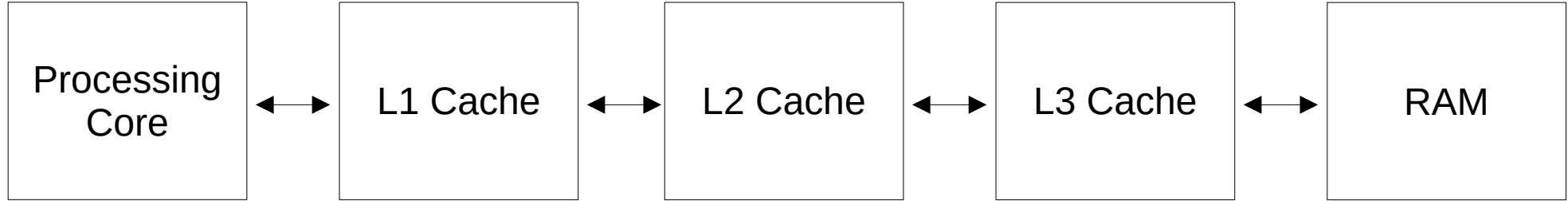
Hey memory system! Give  
me `matrix.at(0).at(2)`!



Hey memory system! Give me `matrix.at(0).at(2)`!

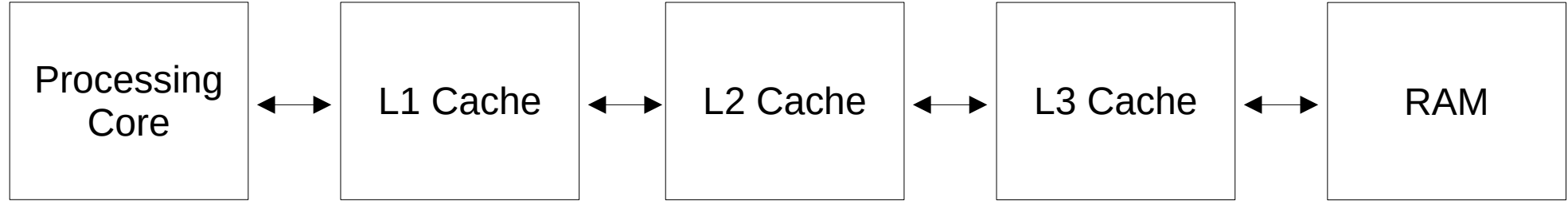
Sure! Its value is 5!

← saves a LOT of time because RAM is slooooooooooooooooooooooow



cache miss →

I don't have that value :(



I'll throw a value out and keep that last one around for next time you need it!

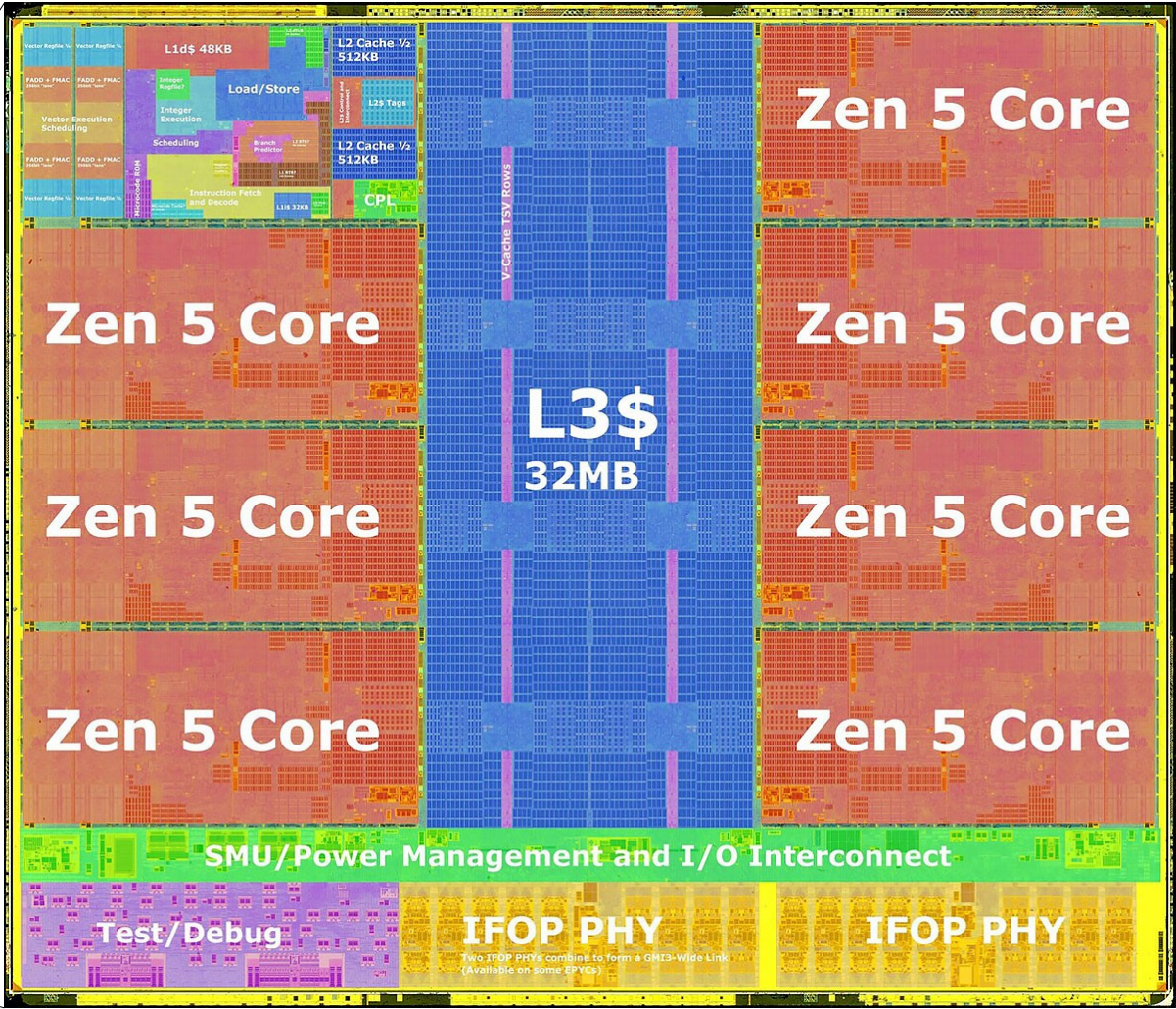
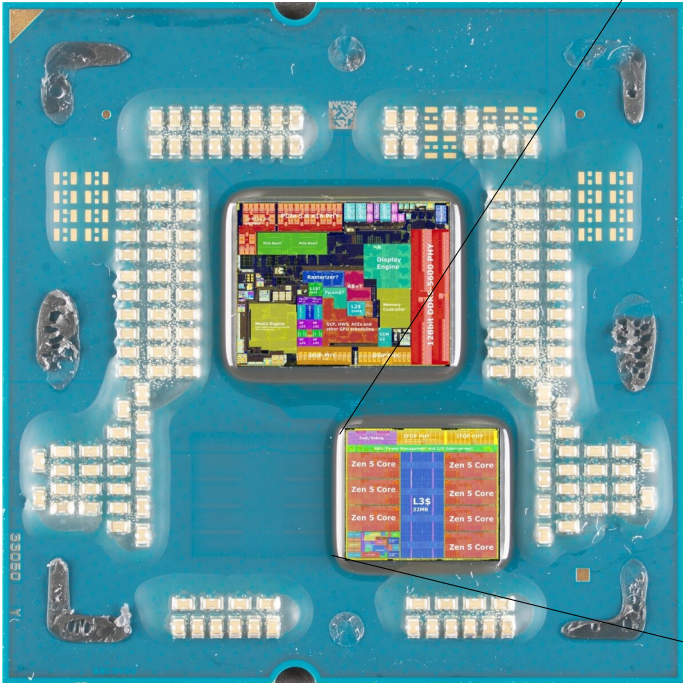
# Cache: levels

- Each level has a higher latency, so the lower we can keep it, the better!

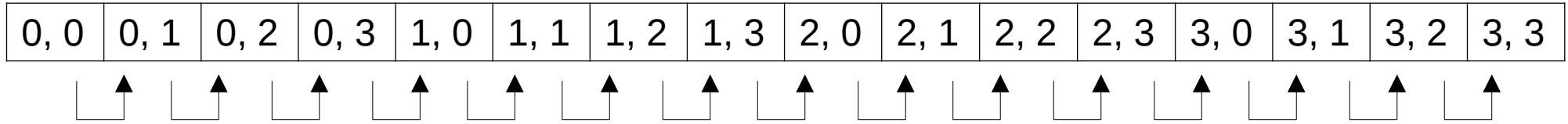
	Approximate latency (cycles)
L1 cache	1
L2 cache	14
L3 cache	50
RAM	350

# AMD Ryzen CCX (5th generation)

- L1 cache: 48 KB (per core)
- L2 cache: 1 MB (per core)
- L3 cache: 32 MB (total)



```
std::vector<std::array<int, 4>> matrix(4);
```



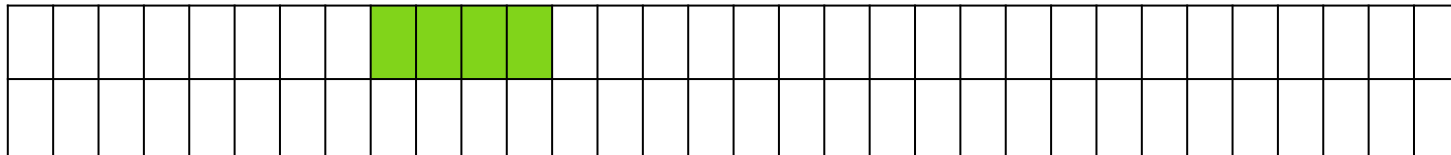
Our previous example processes one element at a time but does not reuse anything.

Why does this still matter?

```
for(int row = 0; row < matrixSize; row++) {  
    for(int col = 0; col < matrixSize; col++) {  
        matrix.at(row).at(col) += 5;  
    }  
}
```

# Cache lines

- Memory systems, and by extension caches, only operate on sequences of (usually 64) bytes
- Even if you only need to read a single byte (char), the processor will fetch the entire cache line from memory.
- Visualised: reading a single float or int



# How to use caches?

- Even though we are iterating over different elements, we are still using the same cache line!



```
std::vector<std::array<int, 4>> matrix(4);
```

0, 0	0, 1	0, 2	0, 3	1, 0	1, 1	1, 2	1, 3	2, 0	2, 1	2, 2	2, 3	3, 0	3, 1	3, 2	3, 3
------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------



We are currently processing row 3, column 1.  
Which element is processed next?

```
for(int row = 0; row < matrixSize; row++) {  
    for(int col = 0; col < matrixSize; col++) {  
        matrix.at(row).at(col) += 5;  
    }  
}
```

# How to use caches?

- Even though we are iterating over different elements, we are still using the same cache line!
- Memory accesses should be predictable (and often are!)
  - **Preferably sequential!**
  - The CPU has a prefetching subsystem that will look ahead in your program, try to guess what memory it will need, and send out memory requests *before* the values are requested by the program.

# Task: which is likely faster (if any)?

```
struct SceneContents {
    vector<array<float, 3>> positions;
    vector<string> names;
    vector<int> actorIDs;
};

int main() {
    const int length = 25000000;
    SceneContents sc;
    sc.positions.resize(length);
    sc.names.resize(length);
    sc.actorIDs.resize(length);
    double sumX = 0;
    for(int i = 0; i < length; i++) {
        sumX += sc.positions.at(i).at(0);
    }
    cout << (sumX / length) << endl;
}
```

```
struct ActorInScene {
    array<float, 3> pos;
    string name;
    int id;
};

int main() {
    const int length = 25000000;
    vector<ActorInScene> actors(length);

    double sumX = 0;
    for(int i = 0; i < length; i++) {
        sumX += actors.at(i).pos.at(0);
    }
    cout << (sumX / length) << endl;
}
```

# Task: which is likely faster (if any)?

```
struct SceneContents {  
    vector<array<float, 3>> positions;  
    vector<string> names;  
    vector<int> actorIDs;  
};  
  
int main() {  
    const int length = 25000000;  
    SceneContents sc;  
    sc.positions.resize(length);  
    sc.names.resize(length);  
    sc.actorIDs.resize(length);  
    double sumX = 0;  
    for(int i = 0; i < length; i++) {  
        sumX += sc.positions.at(i).at(0);  
    }  
    cout << (sumX / length) << endl;  
}
```

**0.69 seconds**

```
struct ActorInScene {  
    array<float, 3> pos;  
    string name;  
    int id;  
};  
  
int main() {  
    const int length = 25000000;  
    vector<ActorInScene> actors(length);  
  
    double sumX = 0;  
    for(int i = 0; i < length; i++) {  
        sumX += actors.at(i).pos.at(0);  
    }  
    cout << (sumX / length) << endl;  
}
```

**0.90 seconds**

# How to use caches?

- Cache systems LOVE sequential memory accesses
  - No waste: we don't need to fetch additional cache lines from RAM
  - Other members in a struct can cause having to “jump over” unused values because structs are still laid out sequentially
    - Solution: “struct of arrays”

# Performance

- How to measure performance
- Tip 1: Use the cache
- **Tip 2: Use a good algorithm**

# Use a good algorithm

- Example: You have a list of names, and would like to check if a person's name is on that list
- You implement a function that iterates over each name in the list and compares it against the one you are looking for.

# Use a good algorithm

- Example: You have a list of names, and would like to check if a person's name is on that list
  - You implement a function that iterates over each name in the list and compares it against the one you are looking for.
  - Better solution: subdivide the list in advance into smaller lists where all names have the same first letter, and only search that one. A bit more work upfront, but quickly starts paying off. Can also be repeated for the second, third, and later characters in each name.

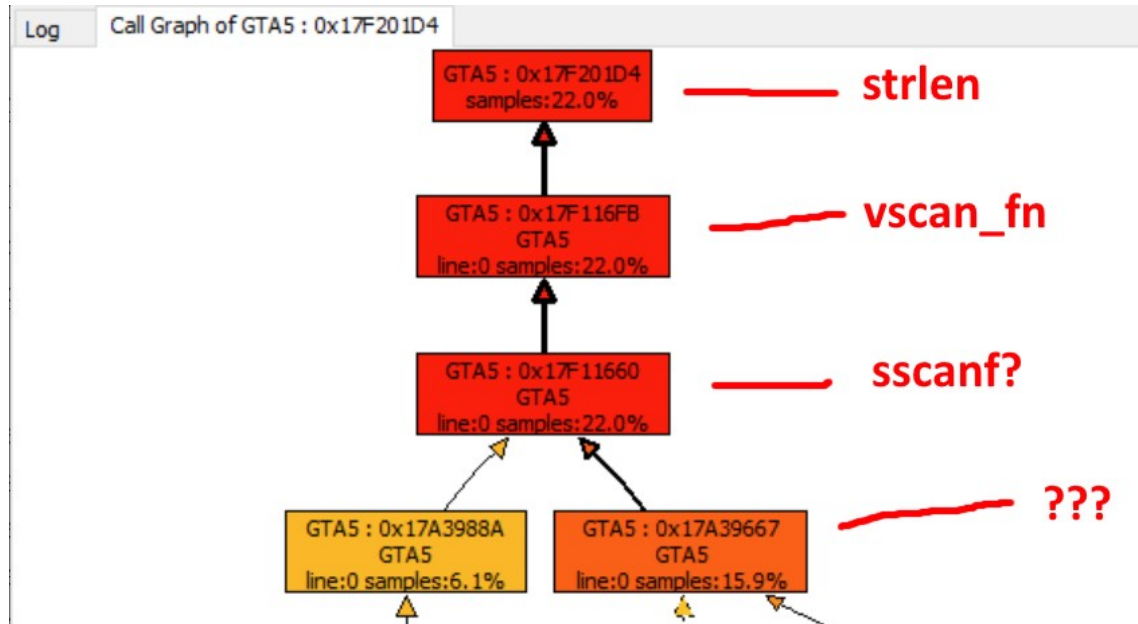


# Use a good algorithm

- Most of this is TDT4120 Algorithms and Data Structures
- Usually the most effective way to speed things up, but also the most difficult.
- One general rule: use `std::unordered_map` when you can, except if you don't have many elements to look through. Then a `std::vector` is probably faster (but always measure!)

# Use a good algorithm: example

- GTA V famously used a terrible parser for loading JSON files, which caused minutes of additional loading time
  - With a small hack someone sped it up by ~70%



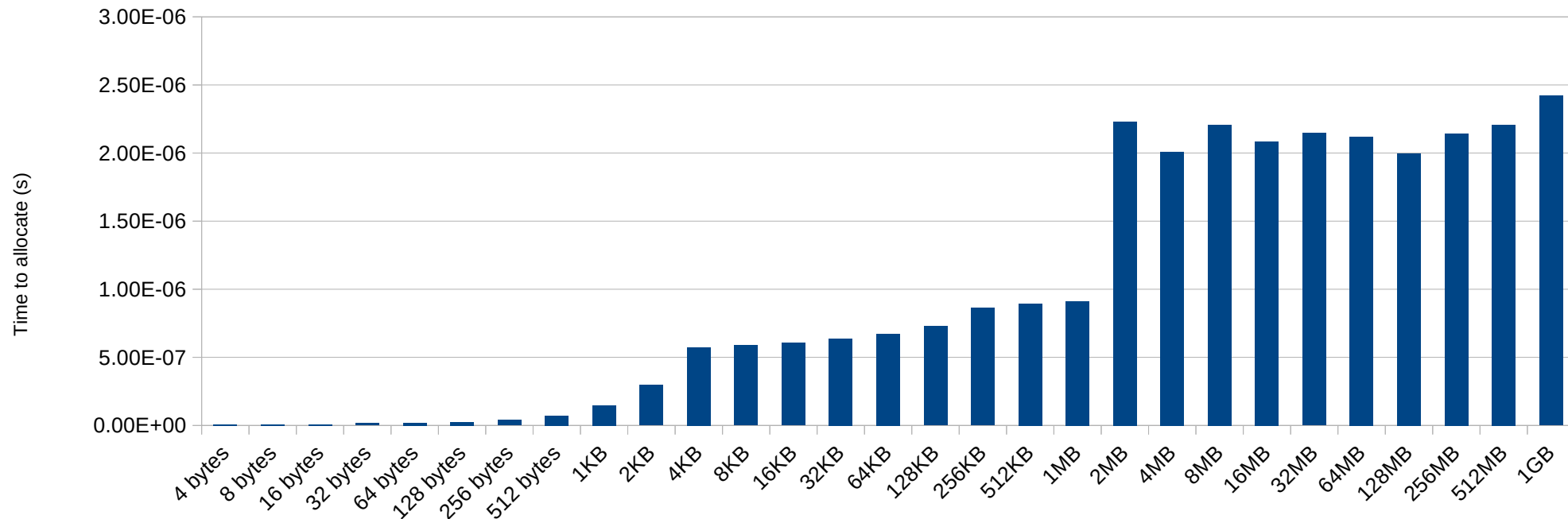
<https://nee.lv/2021/02/28/How-I-cut-GTA-Online-loading-times-by-70/>

# Performance

- How to measure performance
- Tip 1: Use the cache
- Tip 2: Use a good algorithm
- **Tip 3: Follow the allocations**

# Heap allocations

Time taken to allocate and deallocate a specific amount of heap memory



Small allocations like these tend to be the bulk of the number of heap allocations done by a program.

# Heap allocations

- Each allocation costs less than ~6 microseconds.
  - Stack allocations are practically free, but the stack does not have much space
- But: these allocations can occur many times if you're not careful, which can quickly add up!
  - Creating a single string in a section of code that is used all the time explodes its runtime.
  - `std::vector` and `std::string` allocate their contents (mostly) on the heap. **Any time you create a copy of one causes a memory allocation**

# Memory allocations

- What to do:
  - For function parameters, pass vectors and strings by const reference as much as possible
  - We'll look at a trick with `std::vector` later today
  - Avoid using strings as much as possible if you have the choice
    - Try to convert to numbers or enumerations as soon as possible
    - Use a `std::stringstream` for building long strings

# Performance

- How to measure performance
- Tip 1: Use the cache
- Tip 2: Use a good algorithm
- Tip 3: Follow the allocations
- **Tip 4: Enable compiler optimisations**

 Easiest way to get a LOT of speed!

# Enable compiler optimisations

- Compilers can optimise your code automatically

```
4 int a() {  
5     int sum = 0;  
6     for(int i = 0; i < 500; i++) {  
7         sum += i;  
8     }  
9     return sum;  
10 }
```



```
1 a():  
2     str    fp, [sp, #-4]!  
3     add    fp, sp, #0  
4     sub    sp, sp, #12  
5     mov    r3, #0  
6     str    r3, [fp, #-8]  
7     mov    r3, #0  
8     str    r3, [fp, #-12]  
9     b      .L2  
10  
11 .L3:  
12     ldr    r2, [fp, #-8]  
13     ldr    r3, [fp, #-12]  
14     add    r3, r2, r3  
15     str    r3, [fp, #-8]  
16     ldr    r3, [fp, #-12]  
17     add    r3, r3, #1  
18     str    r3, [fp, #-12]  
19  
20 .L2:  
21     ldr    r3, [fp, #-12]  
22     cmp    r3, #500  
23     blt    .L3  
24     ldr    r3, [fp, #-8]  
25     mov    r0, r3  
26     add    sp, fp, #0  
27     ldr    fp, [sp], #4  
28     bx     lr
```



# Enable compiler optimisations

- Compilers can optimise your code automatically
  - Key observation: as long as the program gives the same result, *how it gets there does not matter*

```
4  int a() {  
5      int sum = 0;  
6      for(int i = 0; i < 500; i++) {  
7          sum += i;  
8      }  
9      return sum;  
10 }
```

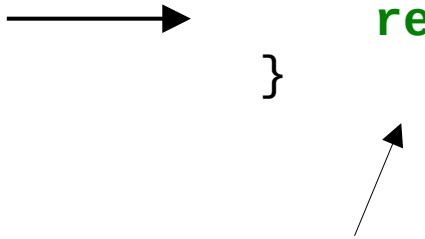


```
1  a():  
2      ldr    r0, .L3  
3      bx    lr  
4  .L3:  
5      .word 124750
```

# Enable compiler optimisations

- Compilers can optimise your code automatically
  - Key observation: as long as the program gives the same result, *how it gets there does not matter*
  - Downside: hard to debug!

```
int a() {  
    int sum = 0;  
    for(int i = 0; i < 500; i++) {  
        sum += i;  
    }  
    return sum;  
}
```



The diagram illustrates a compiler optimization. On the left, a function `a()` is shown with a loop that calculates the sum of integers from 0 to 499. An arrow points to the right, where the same function is shown, but the loop has been replaced by a constant return value of 124750. A second arrow points from the text 'Faster, but no relation anymore to the original program!' to the constant value 124750, highlighting that the compiler has lost track of the original logic.

```
int a() {  
    return 124750;  
}
```

Faster, but no relation anymore  
to the original program!

# Question: issues with measuring

- How can optimisations cause issues with measuring the execution time of a part of a program?

```
int a() {  
    int sum = 0;  
    for(int i = 0; i < 500; i++) {  
        sum += i;  
    }  
    return sum;  
}  
}
```

→

```
int a() {  
    return 124750;  
}
```

# Enable compiler optimisations

- Compilers can optimise your code automatically  
With limits of course, but they're also very smart
- For meson, instead of running:

```
meson setup builddir
```

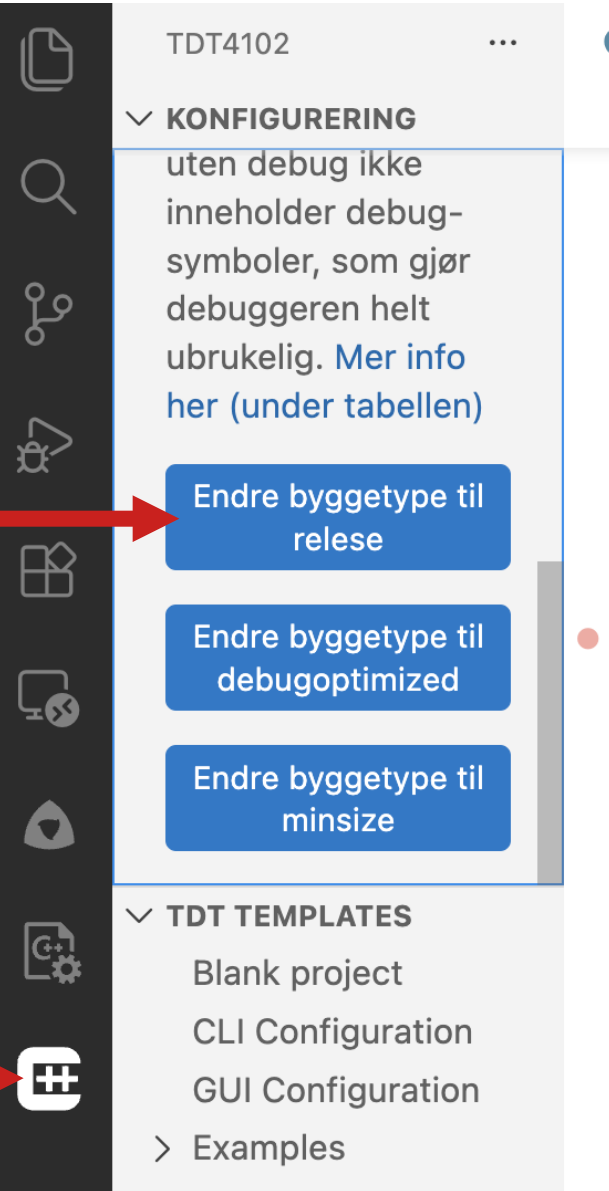
Run it using the command:

```
meson setup builddir -Dbuildtype=release
```

# Enable compiler optimisations

- The extension can also handle this for you :)

2



1



# Enable compiler optimisations

- If you did this correctly, meson will note it in the summary shown at the end:

```
testproject 0.1

Subprojects
  animationwindow: YES

User defined options
  buildtype      : release

Found ninja-1.11.1 at /opt/homebrew/bin/ninja
```

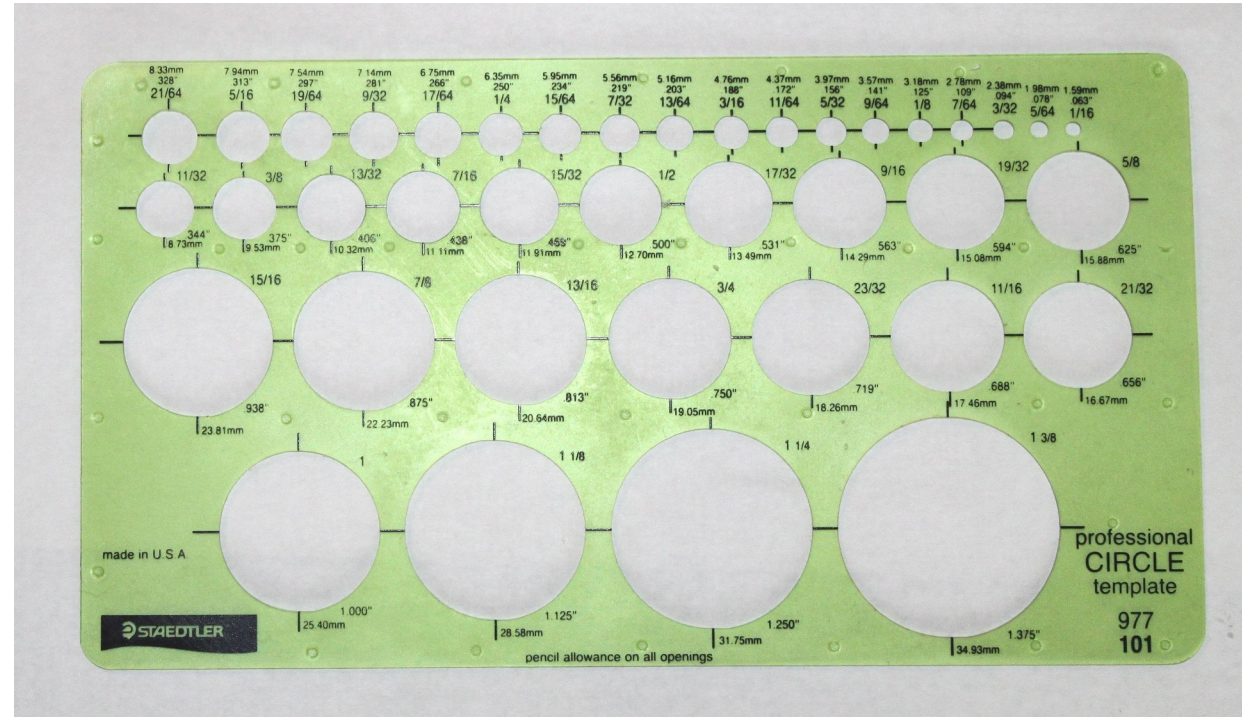
- Only enable optimisations when you know your program works. Debugging with optimisations is rather limited.

# Performance: Summary

- Don't assume something is faster, always measure
- A good algorithm usually gives you the best performance improvements
- Due to the way memory caches work, you should:
  - Iterate over arrays in the order elements are defined
  - Avoid allocating single objects with **new**
- Try to avoid creating copies of vectors and strings, as memory allocations can get quite costly
- Remember to enable compiler optimisations

# Today

- Performance tips
- **Templates**
- More about `std::vector`



Dwight Burdette



# Templates

- Templates are configurable objects or functions
- We have used a LOT of templates already!
- By creating one generic version of an object or function, we can reuse it to work with any data type we choose!

```
std::array<>  
std::vector<>  
std::make_shared<>()  
std::make_unique<>()  
std::map<>  
std::unordered_map<>
```

```
std::pair<>  
std::move<>()  
std::shared_ptr<>  
std::unique_ptr<>  
std::uniform_int_distribution<>  
std::uniform_real_distribution<>
```

# Templates: Syntax

Here we declare a template with parameter T. T is a placeholder that will be replaced by the type we specify when creating an instance:

```
template<typename T>
struct Point {
    T x = 0;
    T y = 0;
};
```

Wherever we used T above is replaced by the one we specify when creating an instance:

```
Point<double> point;
```

```
double Point<double>::x
```

```
point.x = 0;
```

The data type of x and y has been replaced by the one we specified (**double**)

# Templates

- Important: when declaring object templates, you **must** define all methods in the header file.
- You typically don't even make a .cpp file for templated classes and structs

```
// File: Printer.h
#pragma once
#include <iostream>

template<typename T>
struct Printer {
    void print(T value) {
        std::cout << value << std::endl;
    }
};
```

# Templates of objects

- It is possible to create templates of objects and functions:

```
template<typename Type>
class Point {
    Type x = 0;
    Type y = 0;
public:
    void set(Type _x, Type _y) {
        x = _x;
        y = _y;
    }
    Type getX() {
        return x;
    }
    Type getY() {
        return y;
    }
};
```

# Templates of functions

- It is possible to create templates of objects and functions:

```
template<typename Number>
bool isGreater(Number a, Number b) {
    return a > b;
}
```

```
int main() {
    std::cout << isGreater<int>(6, 7) << std::endl;
    return 0;
}
```

▼ We must specify the  
template parameter type  
when calling the function

# Templates

- Templates can have more than one parameter  
(Example: `std::array` and `std::unordered_map`)

```
template<typename Type1, typename Type2>
struct Point {
    Type1 x = 0;
    Type2 y = 0;
};
```

# Templates

- Template parameters can also be values instead of data types
  - Used in `std::array`
  - In practice only used to specify integer types

```
template<typename EntryType, int queueLength>
class Queue {
    std::array<EntryType, queueLength> queue;
public:
    void enqueue(EntryType entry) { /* */ }
    EntryType getNext() { /* */ }
    int getQueueLength() {
        return queueLength;
    }
};
```

# Today

- Performance tips
- Templates
- **More on `std::vector`**



# `std::vector`

- We will now look at how a `std::vector` works
  - Note: `std::string` works very similar as well
- Motivation:
  - `std::vector` is the most used container
  - One line of code can make a `std::vector` much faster

# Task: `std::vector`

- Before we talk about how the vector is implemented, take a moment to consider how you would do it using the techniques you have learned thus far.

How would you implement each of the methods below?

```
template<typename Contents>
class vector {
    /* variables go here */
public:
    int size();
    void push_back(Contents value);
    Contents& at(int index);
    void resize(int newSize);
};
```

# Demonstration: `std::vector`

# std::vector

How to use a vector more effectively:

```
std::vector<int> vec;  
const unsigned long length = 10000000000;
```

```
vec.reserve(length);
```

```
for(unsigned long i = 0; i < length; i++) {  
    vec.push_back(5);  
}
```

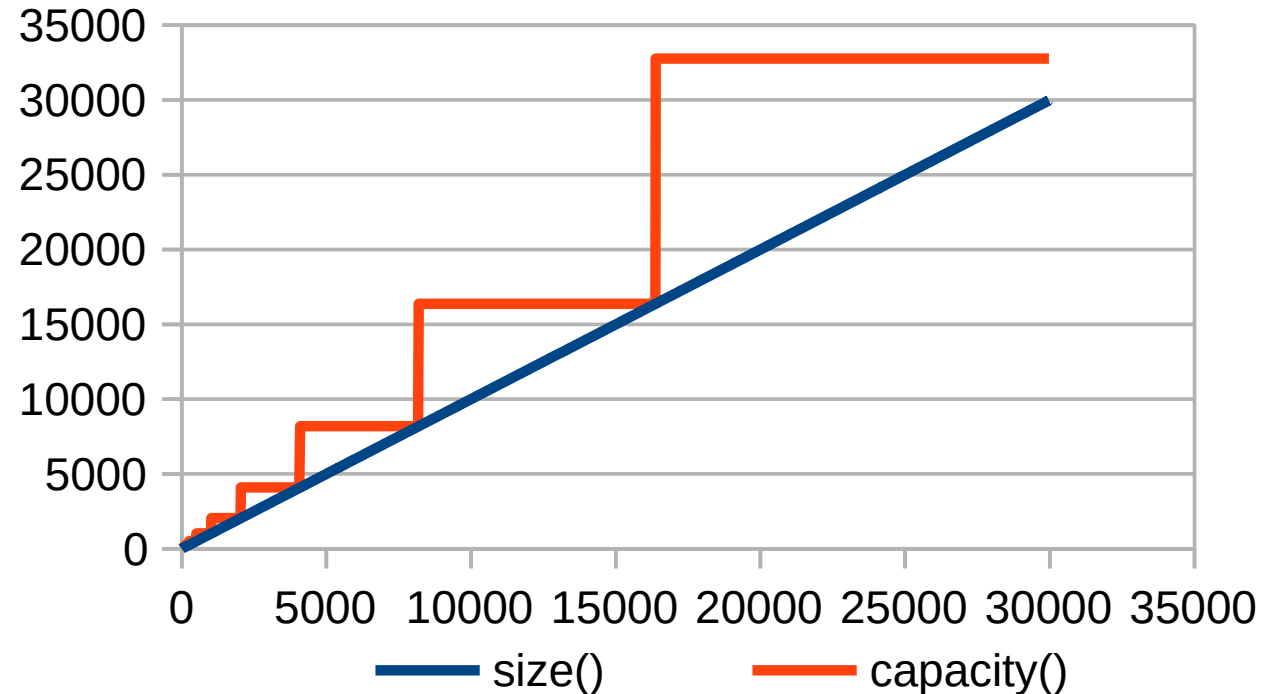
With this line:  
15.8 seconds

Without this line:  
22.9 seconds

std::vector has a capacity() function.

Let's do a little experiment to see what it does!

```
std::vector<double> vec;  
for(int i = 0; i < 30000; i++) {  
    vec.push_back(8);  
    std::cout << vec.size() << ", "  
              << vec.capacity() << std::endl;  
}
```



# std::vector

- What does reserve() do, and why does it matter so much?
- The key lies in how std::vector is implemented, which (may) look something like this:

```
template<typename Type>
class vector {
    std::unique_ptr<Type> array;
    unsigned long size;
    unsigned long capacity;
};
```

← The is the important bit:  
a fixed length array  
allocated on the heap!

# std::vector

- We can only allocate a fixed length array using the **new**[ ] operator.
- We therefore allocate a larger array than we need to store the vector
- Once the array is full, we:
  - Allocate a larger one
  - Copy the vector's contents
  - Delete the old one

# std::vector

Initial:

`vec.size(): 3`

`vec.capacity(): 4`

285	94	74	
-----	----	----	--

`vec.push_back(48):`

`vec.size(): 4`

`vec.capacity(): 4`

We have enough capacity to store the value, so we don't extend the array

285	94	74	48
-----	----	----	----

`vec.push_back(37):`

`vec.size(): 5`

`vec.capacity(): 8`

Our array is full, so we allocate a larger one and copy over the vector's contents

285	94	74	48	37			
-----	----	----	----	----	--	--	--



# `std::vector`

- An overview of useful methods in `std::vector`:
  - `capacity()`:
    - Returns the length of the internal array
  - `reserve(long length)`:
    - Extend the internal array to the specified length
    - Does not change the `size()` of the vector
  - `resize(long length)`:
    - Update both the length of the vector and its internal array
    - If shorter than the vector's `size()`, deletes elements at the end
    - If you want to force a vector to delete its internal array, use this method by calling `resize(0)`

# std::vector

Why is adding `reserve()` faster?

```
std::vector<int> vec;  
const unsigned long length = 10000000000;
```

```
vec.reserve(length);
```

```
for(unsigned long i = 0; i < length; i++) {  
    vec.push_back(5);  
}
```

With this line:  
15.8 seconds

Without this line:  
22.9 seconds

# `std::vector`

- Answer:
  - Allocating memory takes time
  - Copying vector contents takes time
  - We avoid doing both of these by ensuring the internal array has the correct length
- Best practice:
  - If you know the size a `std::vector` is supposed to have in advance, always use `reserve()` or `resize()`, no matter how small the vector is

# Today

- Performance tips
- Templates
- More on `std::vector`
- **Graphical User Interface (GUI)**

# Graphical user interfaces

- A much more familiar way to interact with a program!
- Motivation:
  - Easier to use (as a user) than the terminal
  - Good example of event-driven programming
- Events are handled through «callback» functions
  - We only do something when the user interacts with the interface. Our program is idle otherwise



# User interfaces


- AnimationWindow has different widgets available:

TDT4102::Button



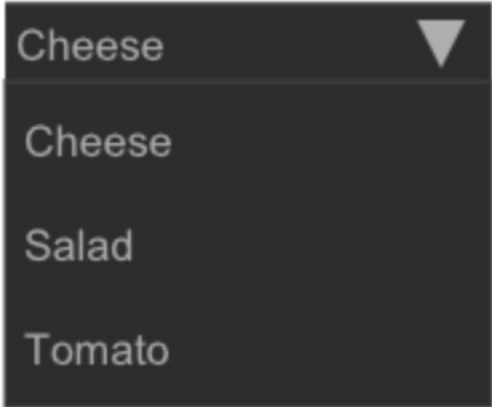
Click me!

TDT4102::TextInput



Text input box

TDT4102::DropDownList



Cheese

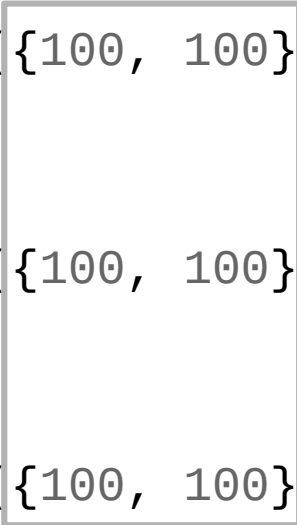


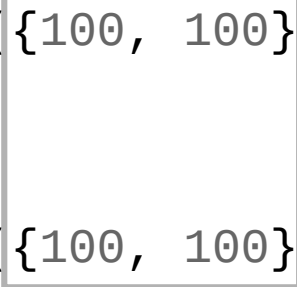





Cheese

Salad

Tomato

# User interfaces

- Creating widgets

	Location	Width	Height	
<code>TDT4102::Button({100, 100},</code>				<code>"Click me!")</code> ← Button label text
<code>TDT4102::TextInput({100, 100},</code>				<code>"Change me!")</code> ← Initial text
<code>TDT4102::DropDownList({100, 100},</code>				<code>dropdownOptions)</code> ← String vector with possible options

# Building an interface

```
void buttonClicked() {  
    std::cout << "Clicked!" << std::endl;  
}
```

A callback function must return void, and have no parameters

```
int main() {  
    TDT4102::AnimationWindow window;  
    TDT4102::Button button({100, 100}, 140, 50, "Continue");  
  
    window.add(button);  
    button.setCallback(&buttonClicked);  
    window.wait_for_close();  
}
```

Adding the widget makes it available in the interface

Use the & operator to get a pointer to the callback function

From here on out we just do stuff when the user does something





# std::bind

- You will need to use a workaround in the assignment to use a member function as a callback function

```
class GUIWindow : TDT4102::AnimationWindow {
    TDT4102::TextInput textInput;

    void textChanged() {
        std::cout << textInput.getText() << std::endl;
    }
public:
    GUIWindow() : textInput({100, 150}, 350, 50, "Change me!") {
        add(textInput);
        textInput.setCallback(std::bind(&GUIWindow::textChanged, this));
    }
};
```

Callback pointer goes here  **this** goes here 

# Today

- Performance tips
- Templates
- More on `std::vector`

# Tomorrow

- Guest lecture on Rust!

# Next week

- Inheritance
- We'll make friends!