

# Lecture 11: Performance on Performance

Bart Iver van Blokland

# We want to hear from you!

- We use questionnaires instead of a reference group
- Extremely important to us for surfacing issues



<https://forms.office.com/e/nFrKFB8xG3>

# Project kickoff: 2 weeks from now

- Think about what you'd like to do
  - Most notable:
    - Must have a graphical user interface (GUI)
    - Must read/write something to a file
    - Must have error handling (later lecture)
    - Must use inheritance (next week's lecture)
- Recommended even if you have enough assignments
- We'll create a question in Piazza later this week for finding someone to cooperate with (if you want to work in pairs)

# From last week

- What does a `std::unique_ptr` and `std::shared_ptr` do?
- What mechanism do they use for their purpose?
- What can a callback function be used for?

The most important thing to remember from today..

# Objective:

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

# Performance

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

# Measuring performance

- Use the Stopwatch class you'll find in the handout code. It uses `std::chrono` to measure time.
- Two problems:
  - Many operations take less than a nanosecond to complete. Clocks in most computers are not accurate enough to measure that time.
  - A number of factors (such as the operating system) add noise to time measurements



# Measuring performance

- Use the Stopwatch class you'll find in the handout code. It uses `std::chrono` to measure time.
- Two problems:
  - Many operations take less than a nanosecond to complete. Clocks in most computers are not accurate enough to measure that time.
  - A number of factors (such as the operating system) add noise to time measurements
- Solution:
  - Run short operations many times
  - Divide the total time taken by the number of repetitions

# Measuring performance

For example:

```
#include "Stopwatch.h"
```

```
int main() {  
    Stopwatch stopwatch;  
    stopwatch.start();  
  
    const long repetitions = 1000000000;  
    for(long i = 0; i < repetitions; i++) {  
        // Operation you want to time goes here  
    }
```

```
    double timeTakenSeconds = stopwatch.stop() / double(repetitions);  
    return 0;  
}
```

# 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;  
    }  
}
```

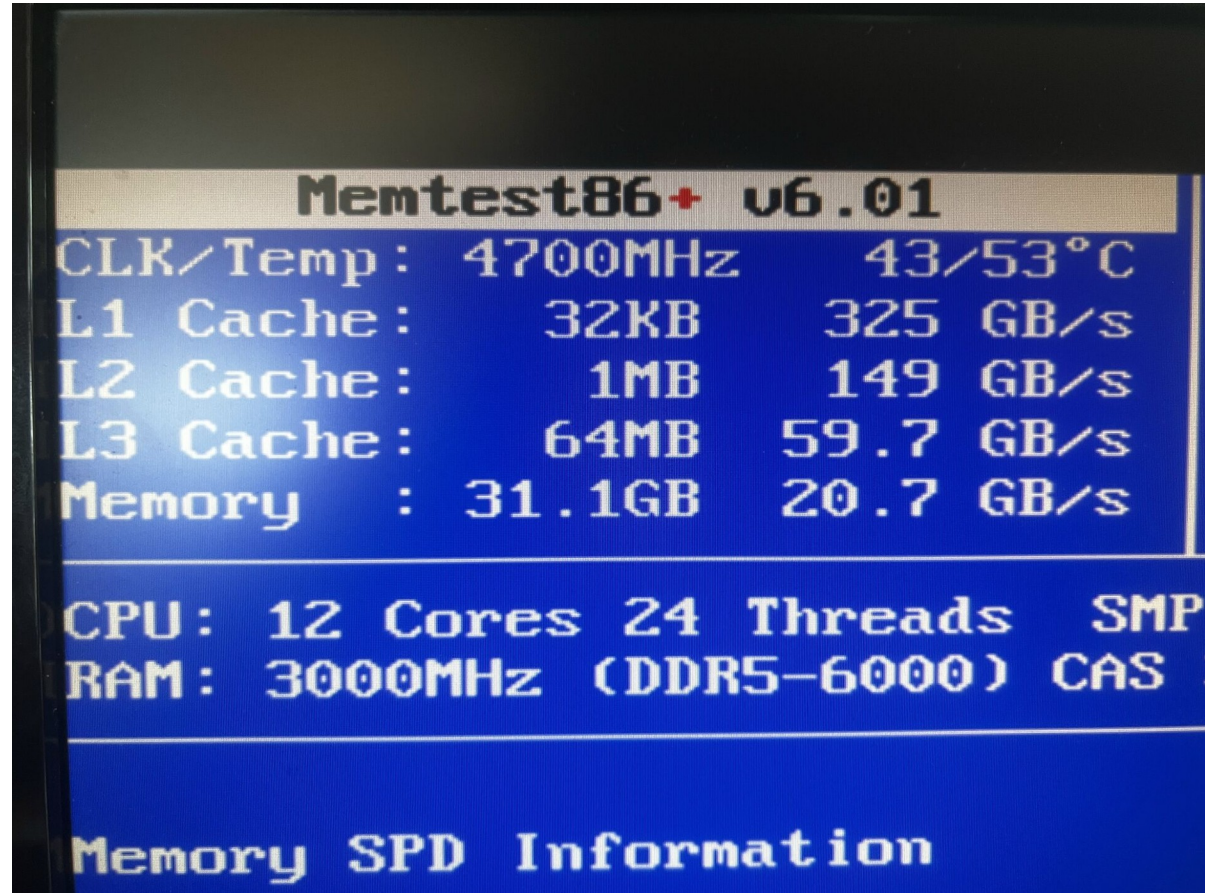
**6.9 seconds**

# Performance

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

# 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

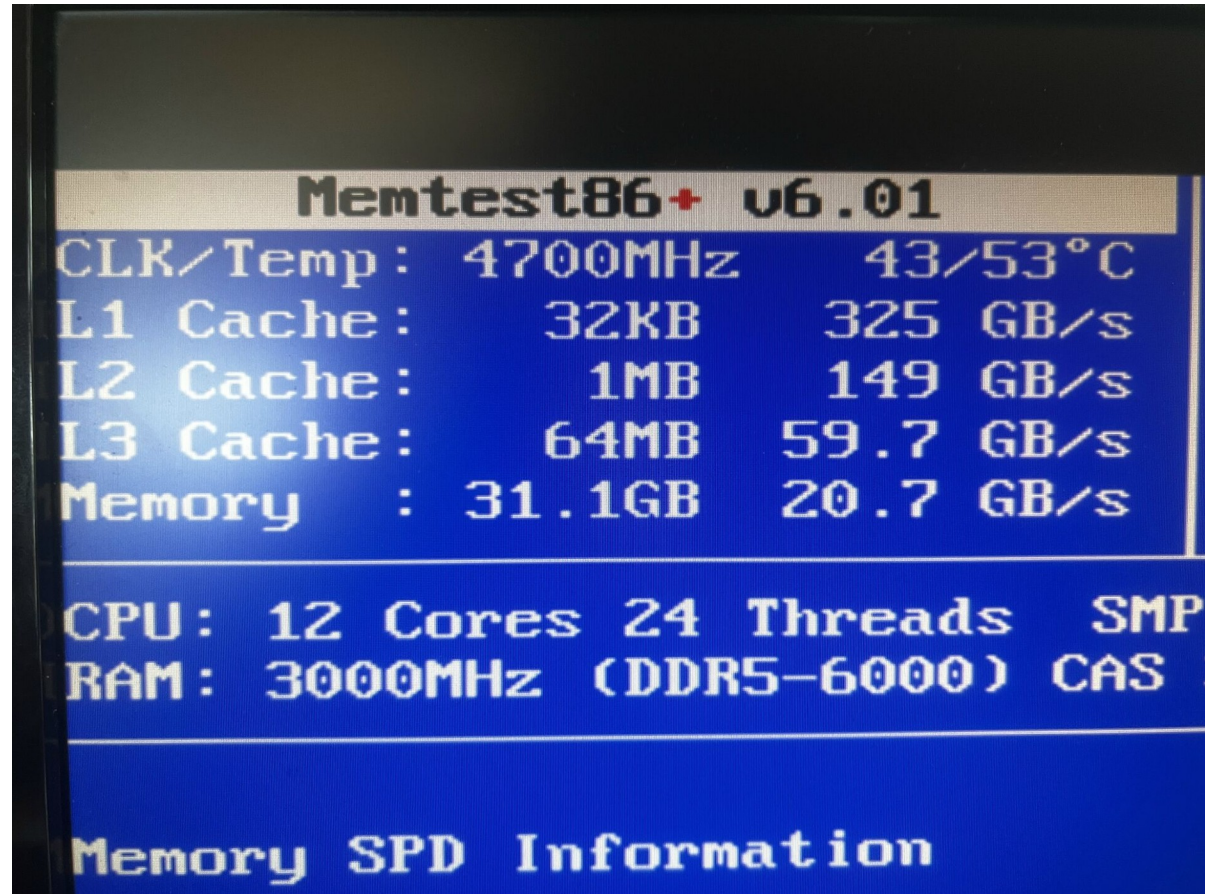


The image shows a screenshot of the Memtest86+ v6.01 boot screen. The screen has a blue background with white text. The title 'Memtest86+ v6.01' is at the top. Below it, system specifications are listed in a table-like format. The specifications include CLK/Temp, L1 Cache, L2 Cache, L3 Cache, Memory, CPU, and RAM. 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 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 in a table-like format. The specifications include CLK/Temp, L1 Cache, L2 Cache, L3 Cache, Memory, CPU, and RAM. 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

- In addition to a difference in bandwidth, there is also a much greater difference in latency between each level

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

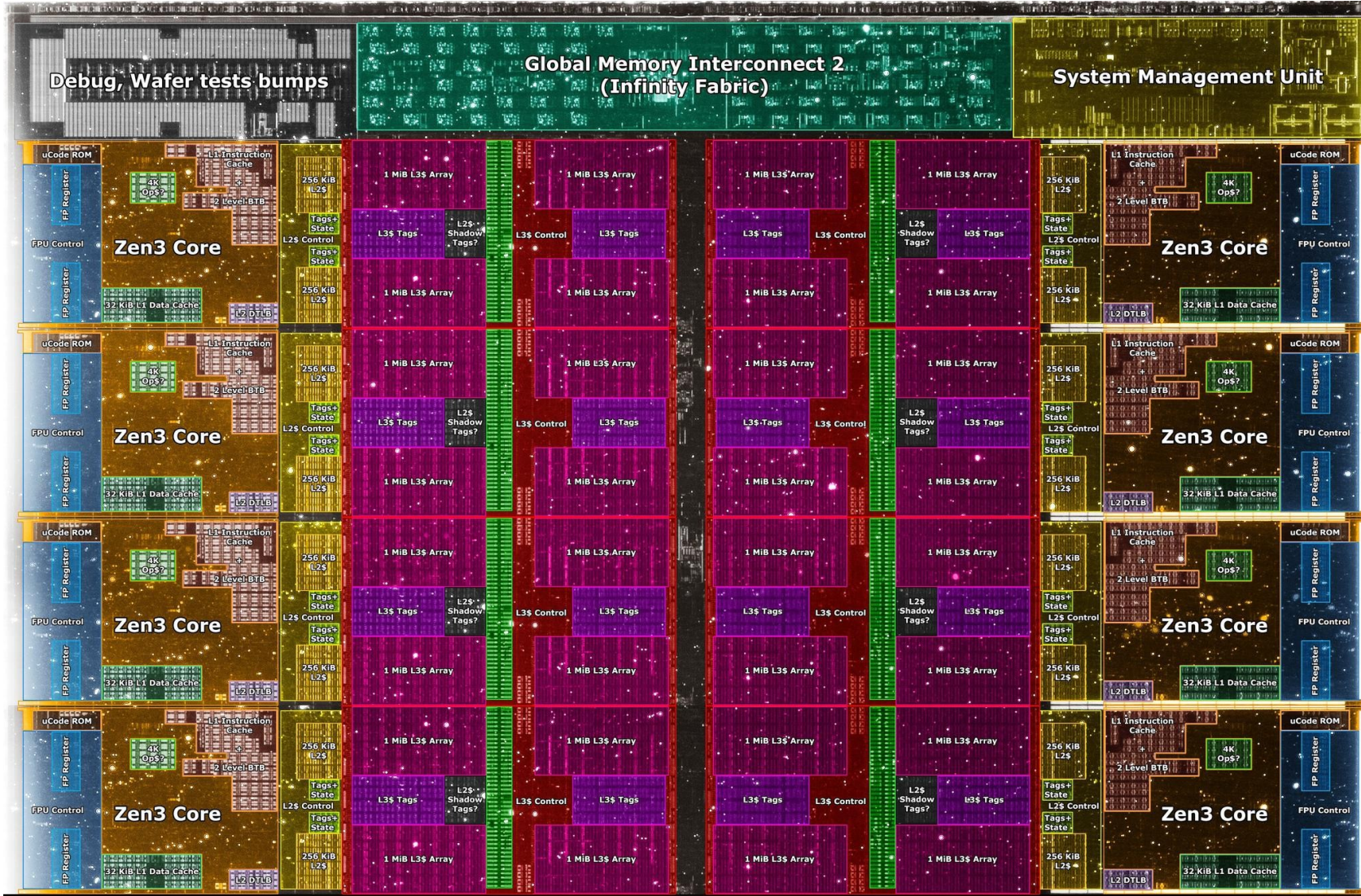


# AMD Ryzen CCX (3rd generation)

L1 cache: 32 KB  
(per core)

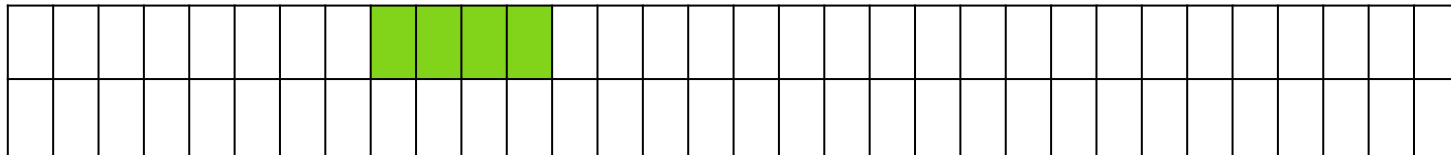
L2 cache: 512 KB  
(per core)

L3 cache: 32 MB  
(total)



# 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



# How to use caches?

- The memory system (including caches) of modern processors are optimised for working with arrays, and iterating over them sequentially
- Avoid arrays of pointers to heap-allocated objects (this will be relevant next week)
- Most processors also have a separate cache for instructions (code).

# 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



# Use a good algorithm

- Example: You have a list of numbers and would like to know the highest number in that list.
- You implement a function that iterates over each number in the list, and for each number compares it against all other numbers in the list. If no number is higher, that is the one you're after.

# Use a good algorithm

- Example: You have a list of numbers and would like to know the highest number in that list.
  - You implement a function that iterates over each number in the list, and for each number compares it against all other numbers in the list. If no number is higher, that is the one you're after.
- Better solution: iterate over each number in the list, and if you find one that is higher than the highest you have seen up to that point, record that as being the maximum

# 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!)

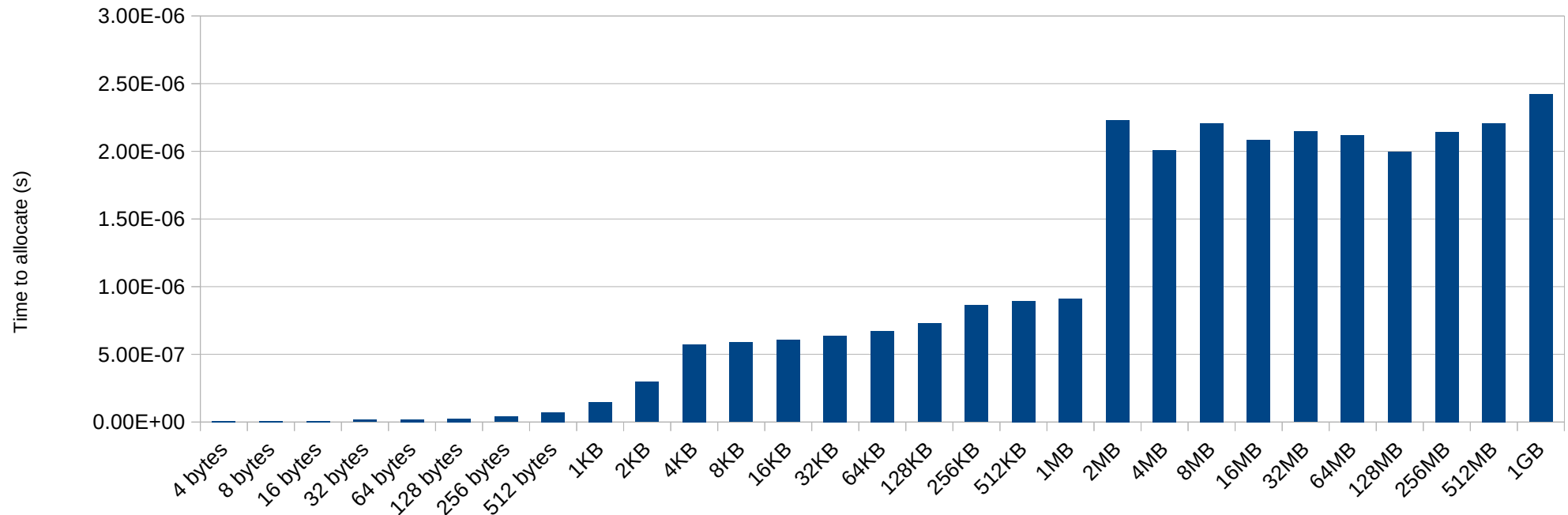
# Performance

- 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



# 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

# 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  
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

- 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
```

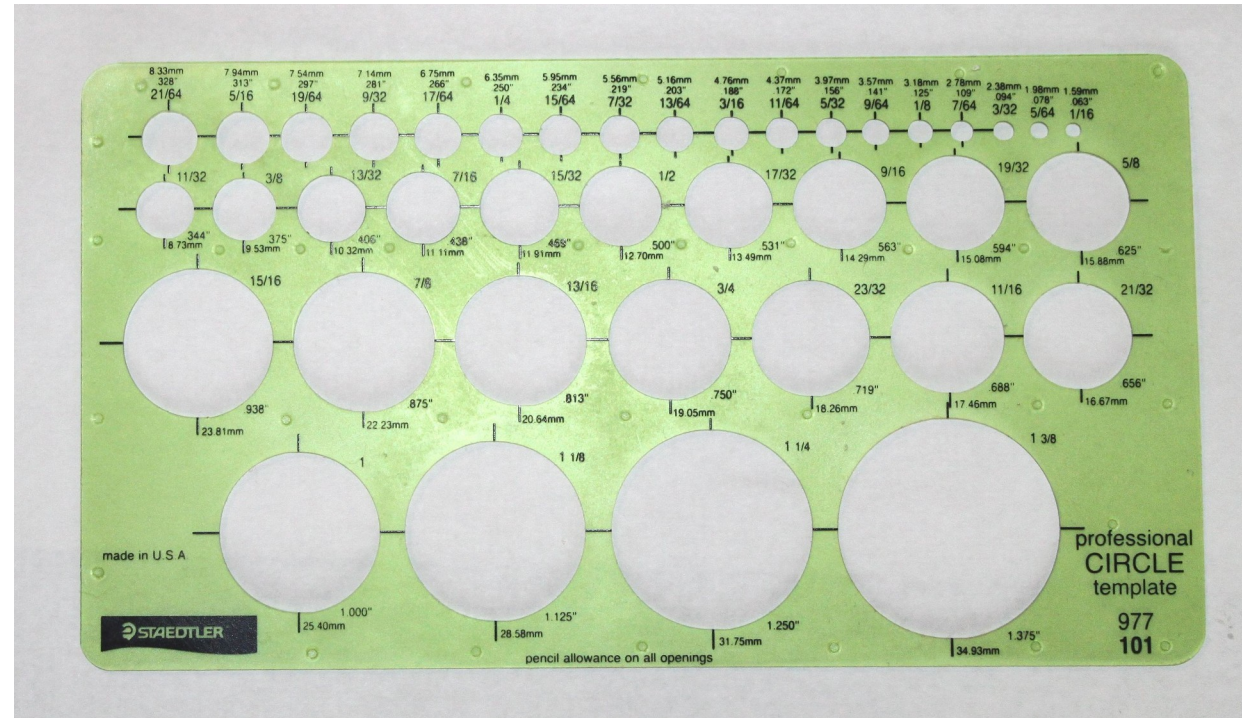
- Only enable optimisations when you know your program works. Debugging with optimisations is rather limited.

# 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
- Motivation:
  - `std::vector` is the most used container
  - One line of code can make a `std::vector` much faster

# 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);  
}
```

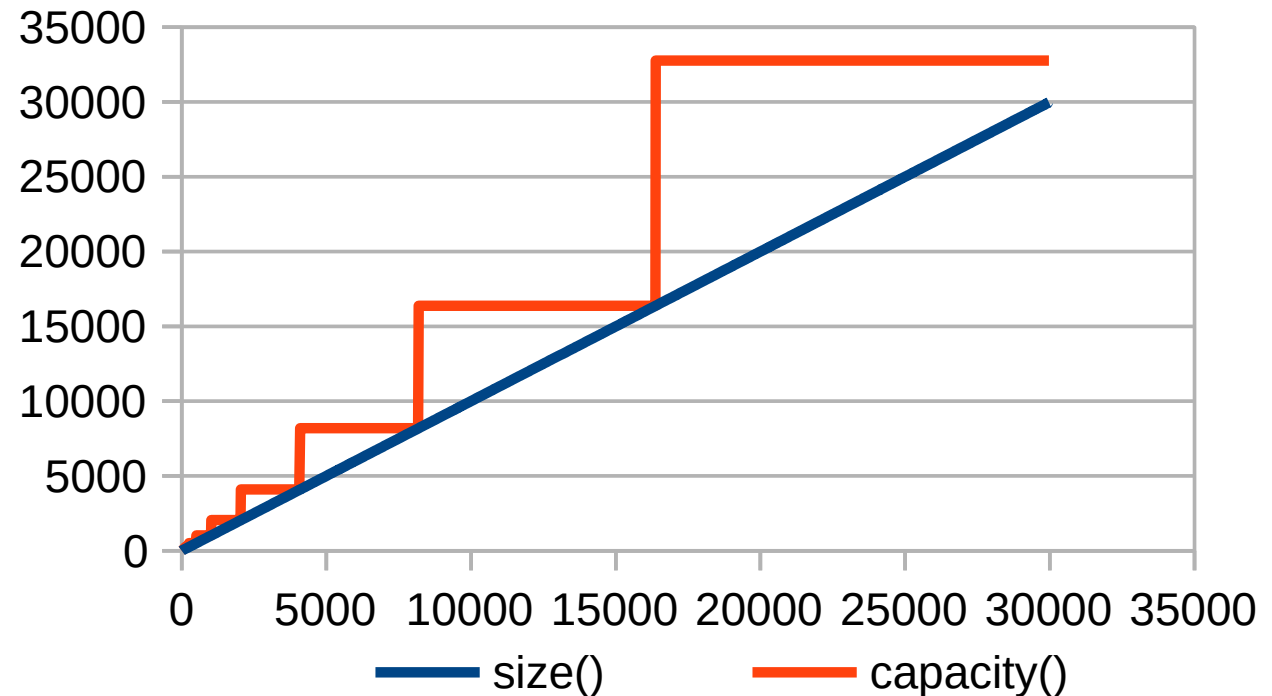
With this line:  
15.8 seconds

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); ←
```

With this line:  
15.8 seconds

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

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`

# Next week

- Inheritance
- We'll make friends!