Optimisation

References

- The sources for information on these slides include:
  - Bulka, D and Mayhew, D. *Efficient C++ : Performance Programming Techniques*.
  - Meyers, S. *Effective C++* and Meyers, S. *More Effective C++.*
  - Sutter, H, *Exceptional C++* and *More Exceptional C++.*
  - My own (often painful) experiences

What is efficiency?

- Depends who you ask.
- Efficiency can be broken down into three main areas:
  - Time efficiency
  - Space efficiency
  - Monetary efficiency
- Usually we mean time efficiency – but often monetary efficiency is more important!

Time Efficiency

- Program must complete a task before a given deadline.
  - The deadline is user and situation defined – it may be acceptable for a scientific calculation to run several hours, but that delay would be inappropriate for real-time audio processing.
- Failure to meet a deadline is critical. Either:
  - Algorithm needs redefinition, or
  - Faster hardware is required, or
  - The deadline must be relaxed.

Space Efficiency

- Program must complete a task within the bounds of available storage.
  - Available storage includes primary storage (RAM) and secondary storage (Hard disk).
- If the task cannot complete within the bounds of the available storage, either:
  - Algorithm needs redefinition, or
  - More storage must be supplied.

Monetary Efficiency

- Program must complete without excessive cost to the user.
- Usually, this is the cost associated with meeting the requirements for time or space efficiency.
- Can include development costs – algorithms may take a lot of skill to implement.

The Space/Time/Money tradeoff

- In most situations, addition of storage requirements will increase time efficiency, while decreasing space efficiency.
- Likewise, reduction of storage requirements will increase space efficiency, but decrease time efficiency.
- Adding storage costs money!
- Example: Cache memory.

C++ & Efficiency

- C++ programs, *if poorly written*, are both time and space inefficient.
- A knowledge of where time inefficiencies occur is vital when programming time-critical systems.

C++ Compared

- A typical C statement translates to somewhere between 5-8 assembly language statements.
- A typical C++ statement translates to somewhere between 5-200 assembly language statements.
- Java? Typically 20 times slower than a C program.
- C++ can be made to be very close to C in terms of performance.

General rules of thumb

- Always pass by reference or by address, never by value.
  - Pass by value makes a copy of the object on the stack – copy constructor call is required, plus destructor call when object goes out of scope at end of function.
- Inline trivial functions to avoid function call overhead.
- Disk and Screen I/O is slow.
- Prefer STL to home-grown code.
- Ensure your algorithms and data structures are optimal.

When to optimize

- “Optimize when a profiler tells you to and not a moment sooner.”, Sutter, *More Exceptional C++.*
- Optimize only when your program/algorithm is complete.
- Optimize only when it is necessary – there is no point spending lots of time optimizing code when it doesn’t need to be optimized.
- If it’s stupid and it works, it’s not stupid. If it meets requirements, then the code is optimal.

An example

- See: opt1.cpp

Analysis

- Debug on: 1101ms, Debug off: 791ms.
- We followed the rules of thumb – pass by reference, inline, etc. We also ensured that we only assigned values when necessary.
- What went wrong? We need to *profile* our code. We use the –pg flag to g++ to allow that.
- *Never think that you can predict where your code is bottlenecked – use a profiler!*

Profile

- With debug on:
  - 64.41% of the time was spent creating or destroying the debug strings
  - 93.23% of the time was spent in string related functions of some kind.
  - 1.69% of the time was spent in user defined functions (testFunc(), main)
- With debug off:
  - 93.23% of the time was spent in string related functions of some kind!

Function calls take time!

- With debug off, we expected very little cost associated with debug.
- Instead, we had over almost all the time taken up by debug code.
- Problem: we were creating string objects, then never using them. This is a Bad Thing™.
- Can we do better? Of course we can…

Improvement Mk I.

- The first problem is that we create a string every time we instantiate the DebugTrace class. This occurs whether debug is on or off.
- Let’s change the string to a string pointer.
- Now we try the same test again.

Analysis

- With debug off, 50% of time is spent on basic housekeeping, and 50% of time is spent in main and testFunc. There is no string overhead, and the cost of DebugTrace objects are nearly zero!
- It takes 40ms to make 1,000,000 testFunc calls.
- With debug on, the percentages are close to the previous version. However, it now takes 1672ms to make 1,000,000 testFunc calls.

Analysis

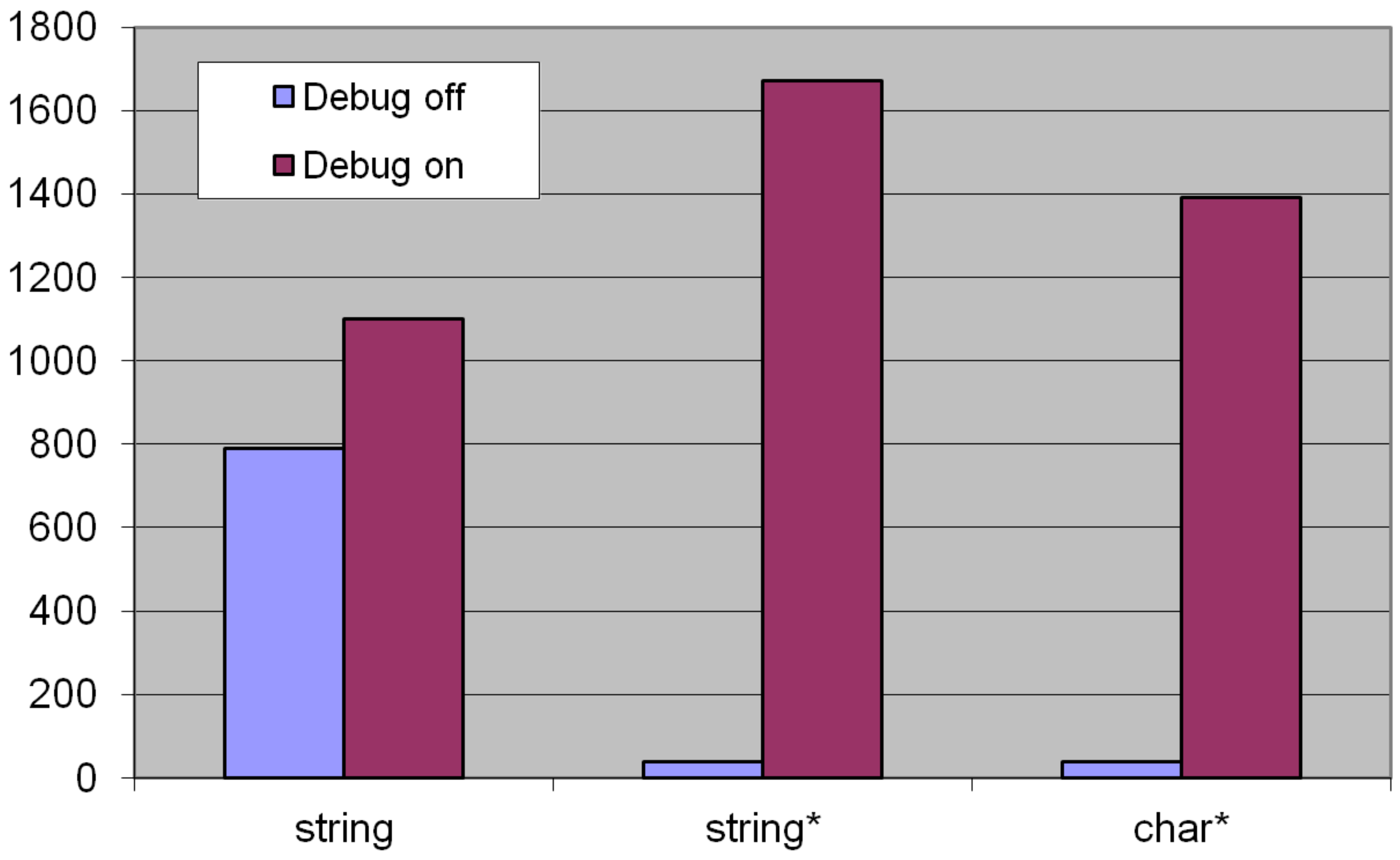
- The extra cost of the debug on case is likely to be OK – performance is not as critical when debugging. Now, we have almost zero overhead for the debug off case.
- Can we improve on this? Yes – we’ll use a `char*` for `s` now.
- Now, we allocate with operator `new[]` and copy with `memcpy`:

Analysis

- Execution time is now 1392ms. 60.53% of time is spent allocating debug messages.
- There will be no real benefit to the “debug off” case.
- Why `memcpy` instead of `strcpy`? `strcpy` looks for a `null` terminator, whereas I knew the size of the string to create the array. By using `memcpy`, we don’t test for the `null`-terminator with each character written to memory.

Results

- Graphically:



The final solution?

- Which was the best solution? In practice, probably the string pointer version, as it would be easier to maintain.
- Readability, reusability and good programming practice are often sacrificed at the altar of efficiency. Ask yourself if it is worth it.

Lessons learnt

- Make sure that objects aren’t being created unnecessarily (such as the string in the first version of the program).
- Ask yourself if you really need the flexibility of a generic solution – if you don’t, then don’t use a generic solution.
- Always use a profiler!

Temporaries

- Temporary objects are a big problem in writing optimal C++.
- Take a look at the following code:

```
// define strings s1 to s4.
// now concatenate those strings
string s = s1 + s2 + s3 + s4;
```

- Firstly, `s1` and `s2` are added together, and a new string created to hold the result. Then, that result and `s3` are added together, and a new string created to hold the result. Finally, that result and `s4` are added together, and the contents copied into string s.

Temporaries

- Two temporary objects are created.
- In total, (including creation of `s1` to `s4`), there have been six constructor calls and one assignment operator call.
- Two unnecessary objects have been created to initialise one! These are temporaries, and these cost time and space.

Temporaries

- To avoid temporaries in the previous situation, try the following:

```
string s1, s2, s3, s4;
string s = s1;
s += s2;
s += s3;
s += s4;
```

- Using the `+=` operator prevents temporaries occurring since we modify string s directly.

More Temporaries

- Using iterators can cause subtle performance issues.

```
for (i=list.begin();i!=list.end();i++)
```

- Under the definition of post-fix increment, a temporary iterator is created and returned for expression evaluation, and then incremented.
- Try instead:

```
for (i=list.begin();i!=list.end();++i)
```

- No temporary is required!

Inlining

- Inlining is the technique of avoiding function call overhead by directly substituting code for functions.
- This is done by the compiler, based on user recommendations. The programmer specifies a function is inline (using the `inline` keyword)
- *It is a recommendation only. The compiler is free to ignore the request.*
- Inline functions equate to C macros, but are type-safe.

When a function is called

- What happens when a function is called depends on the architecture. On most architectures, at least the following must occur:
  - The stack pointer must be saved
  - The return address must be saved
  - A new stack frame allocated for local variables of the function.
  - Any registers that are modified by the function should be saved. There can be a lot of registers to save!
- All of these operations take time, and must occur before the function begins, and must act in reverse when the function ends.

Benefits of Inlining

- Inlining gets rid of the function call overhead by substituting the function directly into the program:

```
inline int myFunc() { return 1; }
 int x = myFunc();
```

- Is directly equivalent to:

```
int x = 1;
```

- Thus, no function call overhead is required – the compiler has made the substitution.
- Inlining contains the single greatest speedup possible, outside of algorithm and data structure change.

Drawbacks of inlining

- Inlining can cause code bloat. Substituting large functions in place of the function call, particularly for functions that are called in a number of separate locations, will result in larger executable sizes.
- Inlining can cause excessive paging of memory to disk, due to a greater likelihood of code spanning pages of memory.
- Larger executable sizes take longer to load, and are likely to run as slow, or slower than the non-inlined version.

When to inline

- Inline trivial functions, accessors, mutators, and *singletons.* A singleton is a function that is only called from one place within a program.
- Inline functions that exist in the *fast path*. The fast path is the part of execution that must be optimal.
- Avoid inlining any large function that is called many times from many different locations within a program – this will cause code bloat.

When to inline

	Large size	Medium Size	Small Size
Low call rate	Don’t inline	Don’t inline	Inline if you have time
Medium call rate	Don’t inline	Rewrite function and inline the fast path	Always inline
High call rate	Rewrite function and inline fast path	Inline if profile data supports	Always inline

Selective inlining

- Inlining in C++ is either all on, or all off.
- *Selective inlining* – inlining based on where in the program the function has been called – can be implemented easily enough.
- See <u>selective-inlining.cpp</u> for an example.

Selective inlining

- The inlined version can be used wherever an inlined version is required or desirable.
- The outlined version is automatically generated by the compiler. Any modifications to the inlined version are reflected in the outlined version – this makes for easy maintenance.

The inline “suggestion”

- The inline keyword is a suggestion to the compiler – it is free to ignore the suggestion if it feels like it.
  - Directly recursive functions are usually not inlined.
  - Polymorphic virtual functions are never inlined.
- There are techniques to avoid these problems, usually by *unrolling* the function, but they make maintenance more difficult, and are probably best left alone.
- Could use templates instead of Polymorphism – see slide 30.

Memory Pools

- Global operator new and delete are comparatively slow.
- Designed to work in multi-threaded environments, so must be thread-safe.
- Thread safing procedures take time, and are not needed for a single-threaded program.
- We can redefine operator new and delete for individual classes to speed up allocation.

Memory Pools

- If operator new is a member function, that version is used for that class only. This allows non-critical classes to still use global new and delete.
- We allocate a fixed size pool of memory, and return blocks from that pool when we call operator new. If we run out of blocks, we enlarge the pool size.

Memory Pools

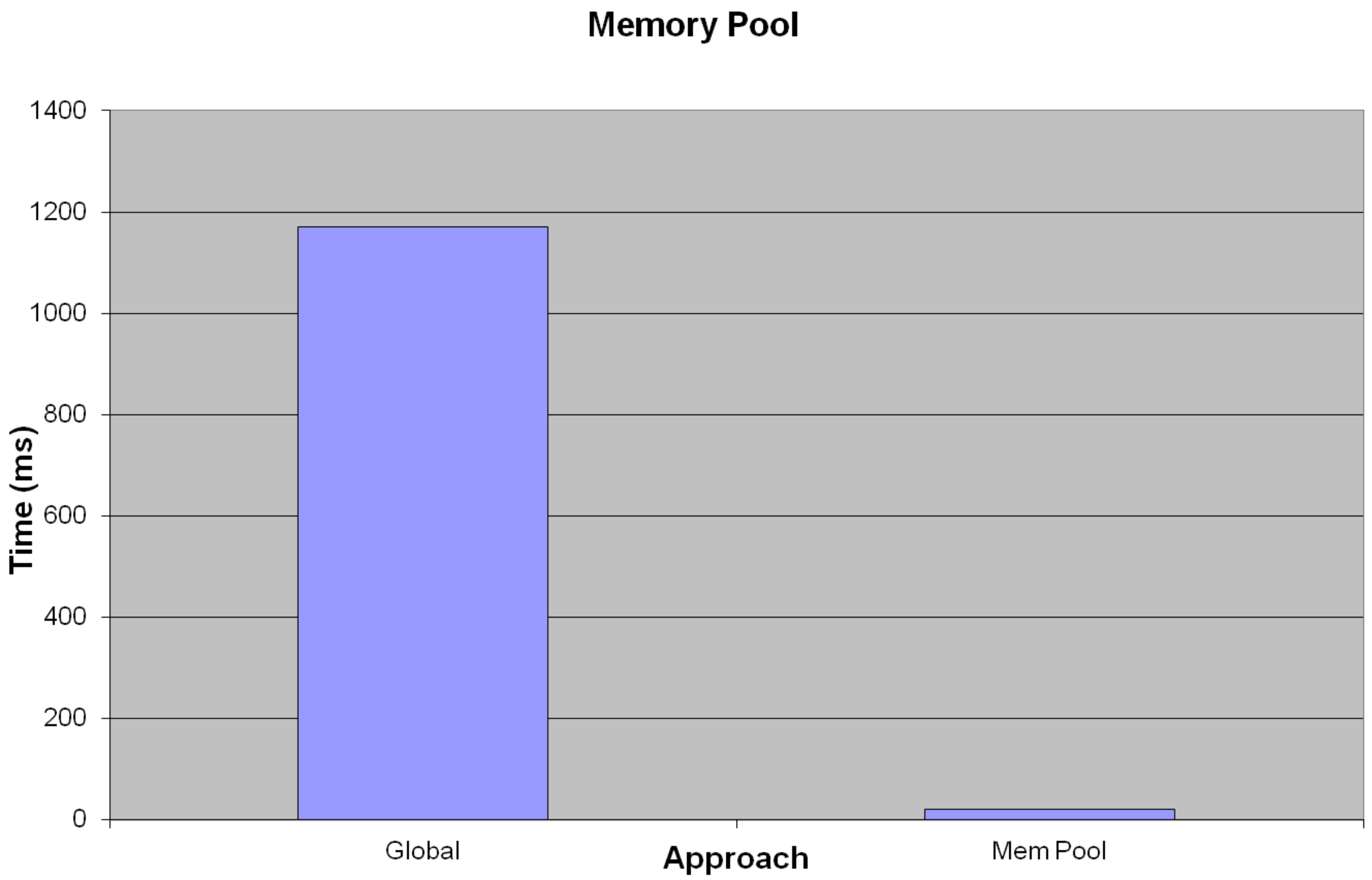
- There is no locking required because we assume this is a single threaded program.
- All memory is allocated in one block by `expandMemoryPool()` whenever the pool runs dry – if you make it large enough, this slight penalty will be small.
- An added benefit: with minor modification, we can perform garbage collection, or easily monitor if we have deleted objects or not.

Testing our implementation

```
yallara.cs.rmit.edu.au% ~/opnew$ g++ -o opnew opnew.cpp -O3 -DUSE_OP_NEW

yallara.cs.rmit.edu.au% ~/opnew$ ./opnew
20 msec

yallara.cs.rmit.edu.au% ~/opnew$ g++ -o opnew opnew.cpp -O3

yallara.cs.rmit.edu.au% ~/opnew$ ./opnew
1171 msec
```

- That’s an 58.55x speed improvement over the global version.

For the visual learners….



Let the compiler do the work

- g++ can do a lot of the simple stuff for you, with a few compile options:
  - -O1: Enable all optimizations that don’t increase compile time
  - -O2: Enable all optimizations that don’t involve space-time tradeoffs.
  - -O3: Enable all optimizations.
  - -Os: Optimize for size. -O2 with a few extra options.
- Most of what you have learnt today *isn’t* done by the compiler, even with –O3.

Where to from here?

- So what happens when all your optimisation results in code that simply isn't fast enough?
- You have several options
  - Check your algorithms and data structures. This is where you will get orders of magnitude improvement. (Bubble sort is still bubble sort!)
  - Avoid advanced features of C++.
  - Get a faster processor, more memory, or whatever you require. This costs money!

Where to from here?

- Your other option completely removes portability…

```
      .file "hello.s"
       .text
    HelloString:
       .asciz "Hello World\n";
       .align 2
    .globl _main
    _main:
       subl $12, %esp
       pushl $HelloString
       call _printf
       movl $0, %eax
       leave
       ret
```

- This is only 422 bytes long – compared to 498 bytes for the compiler generated version, and 482 bytes for the optimised compiler version.
- But it only runs on the x86 architecture.

A final note

- A final quote from Bulka and Mayhew:

"..*the system we were working on suffered from every performance malady in this book, and was finally shot to put it out of its misery…Our contention is that this project…failed because performance was considered a secondary design consideration.* ***He who fails to design for performance, performs a design failure***."