Modern C++ Programming

17. Performance Optimization II

Code Optimization

Federico Busato

2022, v3.15

Table of Context

Table of Context

Table of Context

I/O Operations

I/O Operations

I/O Operations are orders of magnitude slower than memory accesses

I/O Streams

In general, input/output operations are one of the most expensive

- Use `endl` for `ostream` only when it is strictly necessary (prefer `\n`)

- Disable *synchronization* with `printf/scanf`:
  `std::ios_base::sync_with_stdio(false)`

- Disable IO *flushing* when mixing `istream/ostream` calls:
  `<istream_obj>.tie(nullptr);`

- Increase IO *buffer size*:
  `file.rdbuf()->pubsetbuf(buffer_var, buffer_size);`

I/O Streams - Example

```cpp
#include <iostream>

int main() {
    std::ifstream fin;
    // --------------------------------------------------------
    std::ios_base::sync_with_stdio(false); // sync disable
    fin.tie(nullptr);                      // flush disable
                                           // buffer increase
    const int BUFFER_SIZE = 1024 * 1024;   // 1 MB
    char buffer[BUFFER_SIZE];
    fin.rdbuf()->pubsetbuf(buffer, BUFFER_SIZE);
    // --------------------------------------------------------
    fin.open(filename); // Note: open() after optimizations

    // IO operations
    fin.close();
}
```

`printf`

- `printf` is faster than `ostream` (see `speed test link`)
- A `printf` call with a simple format string ending with `\n` is converted to a `puts()` call

```
printf("Hello World\n");
printf("%s\n", string);
```

- No optimization if the string is not ending with `\n` or one or more `%` are detected in the format string

`www.ciselant.de/projects/gcc_printf/gcc_printf.html`

Memory Mapped I/O

A **memory-mapped file** is a segment of virtual memory that has been assigned a direct byte-for-byte correlation with some portion of a file

Benefits:

- Orders of magnitude faster than system calls
- Input can be “cached” in RAM memory (page/file cache)
- A file requires disk access only when a new page boundary is crossed
- Memory-mapping may bypass the page/swap file completely
- Load and store *raw* data (no parsing/conversion)

Memory Mapped I/O - Example


```
#if !defined(__linux__)
    #error It works only on linux
#endif
#include <fcntl.h>          //::open
#include <sys/mman.h>       //::mmap
#include <sys/stat.h>       //::open
#include <sys/types.h>      //::open
#include <unistd.h>         //::lseek
// usage: ./exec <file> <byte_size> <mode>
int main(int argc, char* argv[]) {
   size_t file_size = std::stoll(argv[2]);
   auto   is_read   = std::string(argv[3]) == "READ";
   int fd = is_read ? ::open(argv[1], O_RDONLY) :
                      ::open(argv[1], O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
   if (fd == -1)
       ERROR("::open")                      // try to get the last byte
   if (::lseek(fd, static_cast<off_t>(file_size - 1), SEEK_SET) == -1)
       ERROR("::lseek")
   if (!is_read && ::write(fd, "", 1) != 1) // try to write
       ERROR("::write")
```

Memory Mapped I/O Example


```
auto mm_mode = (is_read) ? PROT_READ : PROT_WRITE;

// Open Memory Mapped file
auto mmap_ptr = static_cast<char*>(
                ::mmap(nullptr, file_size, mm_mode, MAP_SHARED, fd, 0) );

if (mmap_ptr == MAP_FAILED)
    ERROR("::mmap");
// Advise sequential access
if (::madvise(mmap_ptr, file_size, MADV_SEQUENTIAL) == -1)
    ERROR("::madvise");

// MemoryMapped Operations
// read from/write to "mmap_ptr" as a normal array: mmap_ptr[i]

// Close Memory Mapped file
if (::munmap(mmap_ptr, file_size) == -1)
    ERROR("::munmap");
if (::close(fd) == -1)
    ERROR("::close");
```

Consider using optimized (low-level) numeric conversion routines:

```
template<int N, unsigned MUL, int INDEX = 0>
struct fastStringToIntStr;

inline unsigned fastStringToUnsigned(const char* str, int length) {
    switch(length) {
        case 10: return fastStringToIntStr<10, 1000000000>::aux(str);
        case  9: return fastStringToIntStr< 9, 100000000>::aux(str);
        case  8: return fastStringToIntStr< 8, 10000000>::aux(str);
        case  7: return fastStringToIntStr< 7, 1000000>::aux(str);
        case  6: return fastStringToIntStr< 6, 100000>::aux(str);
        case  5: return fastStringToIntStr< 5, 10000>::aux(str);
        case  4: return fastStringToIntStr< 4, 1000>::aux(str);
        case  3: return fastStringToIntStr< 3, 100>::aux(str);
        case  2: return fastStringToIntStr< 2, 10>::aux(str);
        case  1: return fastStringToIntStr< 1, 1>::aux(str);
        default: return 0;
    }
}
```

Low-Level Parsing 2/2

```
template<int N, unsigned MUL, int INDEX>
struct fastStringToIntStr {
    static inline unsigned aux(const char* str) {
        return static_cast<unsigned>(str[INDEX] - '0') * MUL  +
               fastStringToIntStr<N - 1, MUL / 10, INDEX + 1>::aux(str);
    }
};

template<unsigned MUL, int INDEX>
struct fastStringToIntStr<1, MUL, INDEX> {
    static inline unsigned aux(const char* str) {
        return static_cast<unsigned>(str[INDEX] - '0');
    }
};
```

Faster parsing: `lemire.me/blog/tag/simd-swar-parsing`

Speed Up Raw Data Loading

- Hard disk is orders of magnitude slower than RAM
- Parsing is faster than data reading
- Parsing can be avoided by using *binary* storage and `mmap`
- Decreasing the number of hard disk accesses improves the performance $\rightarrow$ **compression**

LZ4 is lossless compression algorithm providing *extremely fast decompression* up to 35% of `memcpy` and good compression ratio
`github.com/lz4/lz4`

Another alternative is **Facebook zstd**
`github.com/facebook/zstd`

Performance comparison of different methods for a file of 4.8 GB of integer values

Load Method	Exec. Time	Speedup
`ifstream`	102 667 ms	1.0x
`memory mapped + parsing (first run)`	30 235 ms	3.4x
`memory mapped + parsing (second run)`	22 509 ms	4.5x
`memory mapped + lz4 (first run)`	3 914 ms	26.2x
`memory mapped + lz4 (second run)`	1 261 ms	81.4x

NOTE: the size of the Lz4 compressed file is 1,8 GB

Memory Optimizations

Heap Memory

- *Dynamic heap allocation is expensive*: implementation dependent and interaction with the operating system
- *Many small heap allocations are more expensive than one large memory allocation*
  The default page size on Linux is 4KB. For smaller/multiple sizes, C++ uses a suballocator
- *Allocations within the page size is faster than larger allocations* (suballocator)

Stack Memory

- *Stack memory is faster than heap memory*. The stack memory provides high locality, it is small (cache fit), and its size is known at compile-time
- `static` stack allocations produce better code. It avoids filling the stack each time the function is reached
- `constexpr` arrays with dynamic indexing produces very inefficient code with GCC. Use `static constexpr` instead

```
void f(int x) {
// bad performance with GCC
//  constexpr        int array[] = {1,2,3,4,5,6,7,8,9};
    static constexpr int array[] = {1,2,3,4,5,6,7,8,9};
    return array[x];
}
```

Memory-Oriented Optimizations

Maximize cache utilization:

- Prefer small data types
- Prefer `std::vector<bool>` over array of `bool`
- Prefer `std::bitset<N>` over `std::vector<bool>` if the data size is known in advance or bounded

Arithmetic

Hardware Notes

- Instruction throughput greatly depends on processor model and characteristics

- Modern processors provide separated units for floating-point computation (`FPU`)

- *Addition*, *subtraction*, and *bitwise operations* are computed by the `ALU` and they have very similar throughput

- In modern processors, *multiplication* and *addition* are computed by the same hardware component for decreasing circuit area $\rightarrow$ multiplication and addition can be fused in a single operation `fma` (floating-point) and `mad` (integer)

uops.info: Latency, Throughput, and Port Usage Information

Data Types

- **32-bit integral vs. floating-point**: in general, integral types are faster, but it depends on the processor characteristics

- **32-bit types are faster than 64-bit types**
  - 64-bit integral types are slightly slower than 32-bit integral types. Modern processors widely support native 64-bit instructions for most operations, otherwise they require multiple operations
  - Single precision floating-points are up to three times faster than double precision floating-points

- **Small integral types are slower than 32-bit integer**, but they require less memory $\rightarrow$ cache/memory efficiency

Operations

- In modern architectures, arithmetic increment/decrement `++` / `--` has the same performance of `add` / `sub`

- **Prefer prefix operator** (`++var`) instead of the postfix operator (`var++`) *

- Use the **compound operators** (`a += b`) instead of operators combined with assignment (`a = a + b`) *

- **Keep near constant values/variables** → the compiler can merge their values

* the compiler automatically applies such optimization whenever possible (this is not ensured for object types)

Integer Multiplication

Integer multiplication requires double the number of bits of the operands

```
// 32-bit PLATFROM

int f1(int x, int y) {
    return x * y; // efficient but can overflow
}

int64_t f2(int64_t x, int64_t y) {
    return x * y; // always correct but slow
}

int64_t f3(int x, int y) {
    return x * static_cast<int64_t>(y); // correct and efficient!!
}
```

Power-of-Two Multiplication/Division/Modulo

- Prefer shift for **power-of-two multiplications** (`a << b`) and **divisions** (`a >> b`) only for run-time values *

- Some **`unsigned`** operations are faster than **`signed`** operations (deal with negative number), e.g. `x / 2`

- Prefer bitwise AND (`a % b` → `a & (b - 1)`) for **power-of-two modulo** operations only for run-time values *

- **Constant multiplication and division** can be heavily optimized by the compiler, even for non-trivial values

---

* the compiler automatically applies such optimizations if `b` is known at compile-time. Bitwise operations make the code harder to read

`Ideal divisors: when a division compiles down to just a multiplication`

Conversion

From	To	Cost
`Signed`	`Unsigned`	no cost, bit representation is the same
`Unsigned`	`Larger Unsigned`	no cost, register extended
`Signed`	`Larger Signed`	1 clock-cycle, register + sign extended
`Integer`	`Floating-point`	4-16 clock-cycles Signed → Floating-point is faster than Unsigned → Floating-point (except `AVX512` instruction set is enabled)
`Floating-point`	`Integer`	fast if `SSE2`, slow otherwise (50-100 clock-cycles)

Floating-Point Division

Multiplication is much faster than division*

not optimized:

```
// "value" is floating-point (dynamic)
for (int i = 0; i < N; i++)
    A[i] = B[i] / value;
```

optimized:

```
div = 1.0 / value;    // div is floating-point
for (int i = 0; i < N; i++)
    A[i] = B[i] * div;
```

* Multiplying by the inverse is not the same as the division
see `lemire.me/blog/2019/03/12`

Floating-Point FMA

Modern processors allow performing `a * b + c` in a single operation, called **fused multiply-add** (`std::fma` in C++11). This implies better performance and accuracy

CPU processors perform computations with a larger register size than the original data type (e.g. 48-bit for 32-bit floating-point) for performing this operation

Compiler behavior:

- GCC 9 and ICC 19 produce a single instruction for `std::fma` and for `a * b + c` with `-O3 -march=native`
- Clang 9 and MSVC 19.* produce a single instruction for `std::fma` but not for `a * b + c`

`FMA: solve quadratic equation`

`FMA: extended precision addition and multiplication by constant`

Compiler Intrinsic Functions

Compiler intrinsics are highly optimized functions directly provided by the compiler instead of external libraries

Advantages:

- Directly mapped to hardware functionalities if available
- Inline expansion
- Do not inhibit high-level optimizations and they are portable contrary to `asm` code

Drawbacks:

- Portability is limited to a specific compiler
- Some intrinsics do not work on all platforms
- The same instricics can be mapped to a non-optimal instruction sequence depending on the compiler

Most compilers provide intrinsics **bit-manipulation functions** for `SSE4.2` or `ABM` (Advanced Bit Manipulation) instruction sets for Intel and AMD processors

GCC examples:

`__builtin_popcount(x)`	count the number of one bits
`__builtin_clz(x)`	(`count leading zeros`) counts the number of zero bits following the most significant one bit
`__builtin_ctz(x)`	(`count trailing zeros`) counts the number of zero bits preceding the least significant one bit
`__builtin_ffs(x)`	(`find first set`) index of the least significant one bit

`gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html`

Compiler Intrinsic Functions

- Compute `integer log2`

```
inline unsigned log2(unsigned x) {
    return 31 - __builtin_clz(x);
}
```

- Check if a number is a `power of 2`

```
inline bool is_power2(unsigned x) {
    return __builtin_popcount(x) == 1;
}
```

- Bit search and clear

```
inline int bit_search_clear(unsigned x) {
    int pos = __builtin_ffs(x); // range [0, 31]
    x      &= ~(1u << pos);
    return pos;
}
```

Example of intrinsic portability issue:

`__builtin_popcount()` GCC produces `__popcountdi2` instruction while Intel Compiler (ICC) produces 13 instructions

`_mm_popcnt_u32` GCC and ICC produce `popcnt` instruction, but it is available only for processor with support for `SSE4.2` instruction set

More advanced usage

- Compute CRC: `_mm_crc32_u32`
- AES cryptography: `_mm256_aesenclast_epi128`
- Hash function: `_mm_sha256msg1_epu32`

software.intel.com/sites/landingpage/IntrinsicsGuide/

Using intrinsic instructions is extremely dangerous if the target processor does not natively support such instructions

Example:

> *"If you run code that uses the intrinsic on hardware that doesn't support the `lzcnt` instruction, the results are unpredictable" - MSVC*

on the contrary, GNU and clang `__builtin_*` instructions are always well-defined. The instruction is translated to a non-optimal operation sequence in the worst case

The instruction set support should be checked at *run-time* (e.g. with `__cpuid` function on MSVC), or, when available, by using compiler-time macro (e.g. `__AVX__`)

Automatic Compiler Function Transformation

`std::abs` can be recognized by the compiler and transformed to a hardware instruction

In a similar way, C++20 provides a portable and efficient way to express bit operations `<bit>`

rotate left	:	`std::rotl`
rotate right	:	`std::rotr`
count leading zero	:	`std::countl_zero`
count leading one	:	`std::countl_one`
count trailing zero	:	`std::countr_zero`
count trailing one	:	`std::countr_one`
population count	:	`std::popcount`

Why is the standard "abs" function faster than mine?

Value in a Range

Checking if a non-negative value *x* is within a range *[A, B]* can be optimized if *B* > *A* (useful when the condition is repeated multiple times)

```
if (x >= A && x <= B)

// STEP 1: subtract A
if (x - A >= A - A && x - A <= B - A)
// -->
if (x - A >= 0 && x - A <= B - A) // B - A is precomputed

// STEP 2
//    - convert "x - A >= 0" --> (unsigned) (x - A)
//    - "B - A" is always positive
if ((unsigned) (x - A) <= (unsigned) (B - A))
```

Value in a Range Examples

Check if a value is an uppercase letter:

```
uint8_t x = ...
if (x >= 'A' && x <= 'Z')
    ...
```

→

```
uint8_t x = ...
if (x - 'A' <= 'Z')
    ...
```

A more general case:

```
int x = ...
if (x >= -10 && x <= 30)
    ...
```

→

```
int x = ...
if ((unsigned) (x + 10) <= 40)
    ...
```

The compiler applies this optimization only in some cases (tested with GCC/Clang 9 -O3)

Lookup Table

Lookup table (LUT) is a *memoization* technique which allows replacing *runtime* computation with <u>precomputed</u> values

Example: a function that computes the logarithm base 10 of a number in the range [1-100]

```
template<int SIZE, typename Lambda>
constexpr std::array<float, SIZE> build(Lambda lambda) {
    std::array<float, SIZE> array{};
    for (int i = 0; i < SIZE; i++)
        array[i] = lambda(i);
    return array;
}
float log10(int value) {
    constexpr auto lamba = [](int i) { return std::log10f((float) i); };
    static constexpr auto table = build<100>(lambda);
    return table[value];
}
```

Low-Level Optimizations

Collection of low-level implementations/optimization of common operations:

- **Bit Twiddling Hacks**
  graphics.stanford.edu/∼seander/bithacks.html

- **The Aggregate Magic Algorithms**
  aggregate.org/MAGIC

- **Hackers Delight Book**
  www.hackersdelight.org

Low-Level Information

The same instruction/operation may take different clock-cycles on different architectures/CPU type

- **Agner Fog - Instruction tables** (latencies, throughputs)
  `www.agner.org/optimize/instruction_tables.pdf`
- **Latency, Throughput, and Port Usage Information**
  `uops.info/table.html`

Control Flow

Branches are expensive 1/2

Computation is faster than decision

Branches are expensive 2/2

Pipelines are an essential element in modern processors. Some processors have up to 20 pipeline stages (14/16 typically)

The downside to long pipelines includes the danger of **pipeline stalls** that waste CPU time, and the time it takes to reload the pipeline on **conditional branch** operations (`if`, `while`, `for`)

Control Flow

- Prefer `switch` statements instead of multiple `if`
  - If the compiler does not use a jump-table, the cases are evaluated in order of appearance → the most frequent cases should be placed before
  - Some compilers (e.g. `clang`) are able to translate a sequence of `if` into a `switch`
- Prefer **square brackets** syntax `[]` over pointer arithmetic operations for array access to facilitate compiler loop optimizations (polyhedral loop transformations)
- Prefer **signed integer** for **loop indexing**. The compiler optimizes more aggressively such loops since integer overflow is not defined
- Prefer range-based loop for iterating over a container [1]

[1] The Little Things: Everyday efficiencies

Control Flow

- In general, `if` statements affect performance when the branch is taken
- Some compilers (e.g. `clang`) use assertion for optimization purposes: most likely code path, not possible values, etc. [2]
- Not all control flow instructions (or branches) are translated into `jump` instructions. If the code in the branch is small, the compiler could optimize it in a conditional instruction, e.g. `ccmovl`
  Small code section can be optimized in different ways [3] (see next slides)

---

[1] Branch predictor: How many 'if's are too many?

[2] Andrei Alexandrescu

[3] Is this a branch?

Minimize Branch Overhead

- **Branch prediction**: technique to guess which way a branch takes. It requires hardware support and it is generically based on dynamic history of code executing
- **Branch predication**: a conditional branch is substituted by a sequence of instructions from both paths of the branch. Only the instructions associated to a *predicate* (boolean value), that represents the direction of the branch, are actually executed

```
int x = (condition) ? A[i] : B[i];
P = (condition) // P: predicate
@P  x = A[i];
@!P x = B[i];
```

- **Speculative execution**: execute both sides of the conditional branch to better utilize the computer resources and commit the results associated to the branch taken

Loop Hoisting

Loop Hoisting, also called *loop-invariant code motion*, consists of moving statements or expressions outside the body of a loop *without affecting the semantics* of the program

Base case:

```
for (int i = 0; i < 100; i++)
    a[i] = x + y;
```

Better:

```
v = x + y;
for (int i = 0; i < 100; i++)
    a[i] = v;
```

Loop hoisting is also important in the evaluation of loop conditions

Base case:

```
// "x" never changes
for (int i = 0; i < f(x); i++)
   a[i] = y;
```

Better:

```
int limit = f(x);
for (int i = 0; i < limit; i++)
   a[i] = y;
```

In the worst case, `f(x)` is evaluated at every iteration (especially when it belongs to another translation unit)

Loop unrolling (or **unwinding**) is a loop transformation technique which optimizes the code by removing (or reducing) loop iterations

The optimization produces better code at the expense of binary size

Example:

```
for (int i = 0; i < N; i++)
    sum += A[i];
```

can be rewritten as:

```
for (int i = 0; i < N; i += 8) {
    sum += A[i];
    sum += A[i + 1];
    sum += A[i + 2];
    sum += A[i + 3];
    ...
} // we suppose N is a multiple of 8
```

Loop unrolling notes:

- + Improve instruction-level parallelism (ILP)
- + Allow vector (SIMD) instructions
- + Reduce control instructions and branches
- - Increase compile-time/binary size
- - Require more instruction decoding
- - Use more memory and instruction cache

Unroll directive The `Intel`, `IBM`, and `clang` compilers (but not `GCC`) provide the preprocessing directive `#pragma unroll` (to insert above the loop) to force loop unrolling. The compiler already applies the optimization in most cases

`lemire.me/blog/2019/04/12`

Branch Hints

C++20 `[[likely]]` and `[[unlikely]]` provide a hint to the compiler to optimize a conditional statement, such as `while`, `for`, `if`

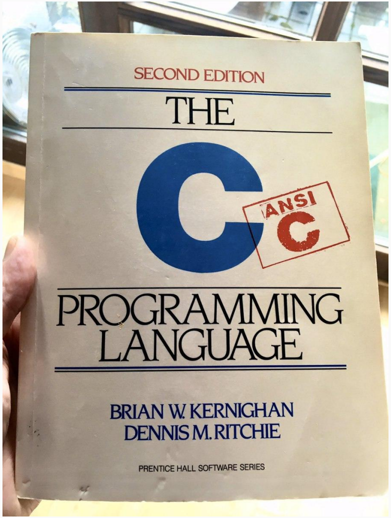
```
for (i = 0; i < 300; i++) {
    [[unlikely]] if (rand() < 10)
        return false;
}
```

```
switch (value) {
  [[likely]]   case 'A': return 2;
  [[unlikely]] case 'B': return 4;
}
```

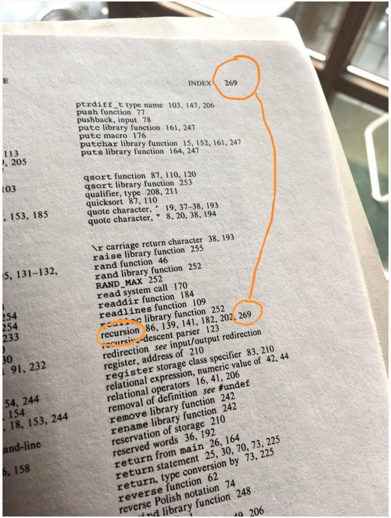
Avoid run-time recursion (very expensive). Prefer *iterative* algorithms instead (see next slides)

Recursion cost: The program must store all variables (snapshot) at each recursion iteration on the stack, and remove them when the control return to the caller instance

The **tail recursion** optimization avoids to maintain caller stack and pass the control to the next iteration. The optimization is possible only if all computation can be executed before the recursive call



SECOND EDITION
THE
C
ANSI
C
PROGRAMMING
LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PRENTICE HALL SOFTWARE SERIES



INDEX 269
ptrdiff_t type name 103, 147, 206
push function 77
pushback, input 78
putc library function 161, 247
putc macro 176
putchar library function 15, 152, 161, 247
puts library function 164, 247
qsort function 87, 110, 120
qsort library function 253
qualifier, type 208, 211
quicksort 87, 110
quote character, ' 19, 37–38, 193
quote character, " 8, 20, 38, 194
\r carriage return character 38, 193
raise library function 255
rand function 46
rand library function 252
RAND_MAX 252
read system call 170
readdir function 184
readlines function 109
library function 252
recursion 86, 139, 141, 182, 202, 269
descent parser 123
redirection see input/output redirection
register, address of 210
register storage class specifier 83, 210
relational expression, numeric value of 42, 44
relational operators 16, 41, 206
removal of definition see #undef
remove library function 242
rename library function 242
reservation of storage 210
reserved words 36, 192
return from main 26, 164
return statement 25, 30, 70, 73, 225
return, type conversion by 73, 225
reverse function 62
reverse Polish notation 74
library function 248

Functions

Function call methods:

Direct	Function address is known at compile-time
Indirect	Function address is known only at run-time
Inline	The function code is fused in the caller code

Function call cost:

- The caller pushes the arguments on the stack in reverse order
- Jump to function address
- The caller clears (pop) the stack
- The function pushes the return value on the stack
- Jump to the caller address

pass by-value small data types ($\leq 8/16$ bytes)
The data are copied into registers, instead of stack

pass by-pointer introduces one level of indirection
They should be used only for raw pointers (potentially `NULL`)

pass by-reference *may* introduce one level of indirection
`pass-by-reference` is more efficient than `pass-by-pointer` as it facilitates variable elimination by the compiler, and the function code does not require checking for `NULL` pointer

Most compilers optimize **pass by-value** with **pass by-reference** for *passive* data structures

---

`Three reasons to pass std::string_view by value`

Argument Passing

For *active* objects with non-trivial (and expensive) copy constructor or destructor:

by-value Expensive, hard to optimize

by-pointer/reference ok. Prefer pass-by-`const`-X (`const` overloading can also be cheaper)

For *passive* objects with trivial copy constructor *and* destructor:

by-const-value Always produce the optimal code (converted in `pass-by-const ref` if needed) but it should be avoided for as it does not change the function signature

by-value Produce optimal code except for GCC (tested with GCC 9.2)

by-reference Could introduce a level of indirection

Function Optimizations

- *Pass by-value built-in types and passive data structured (no side-effect*. The compiler already applies heuristics to determine the most efficient way to pass the parameter (`by-value` or `by-reference`). Pass by-reference does not allow the compiler to optimize in pass `by-value` (if not inline)

- *Keep small the number of function parameters*. The parameters can be passed by using the registers instead filling and emptying the stack

- Consider *combining several function parameters* in a structure

- `const` modifier applied to pointers and references *does not produce better code* in most cases, but it is useful for ensuring read-only accesses

GoTW#81

inline (internal linkage)

`inline` specifier when applied to internal linkage functions (`static` or `anonymous namespace`) is a hint for the compiler.

The code of the function can be copied where it is called (*inlining*)

```
inline void f() { ... }
```

- It is just a hint for the compiler that can ignore it (`inline` increases the compiler heuristic threshold)
- `inline` functions increase the binary size because they are expanded in-place for every function call

Compilers have different heuristics for function inlining

- Number of lines (even comments: `How new-lines affect the Linux kernel performance`)
- Number of assembly instructions
- Inlining depth (recursive)

GCC/Clang extensions allow to *force* inline/non-inline functions:

```
__attribute__((always_inline)) void f() { ... }
__attribute__((noinline))      void f() { ... }
```

- `An Inline Function is As Fast As a Macro`
- `Inlining Decisions in Visual Studio`

Local Functions

All compilers, except MSVC, export all function symbols → slow, the symbols can be used in other translation units

Alternatives:

- Use `static` functions
- Use `anonymous namespace` (functions and classes)
- Use GNU extension (also `clang`) `__attribute__((visibility("hidden")))`

gcc.gnu.org/wiki/Visibility

Consider the following example:

```
// suppose f() is not inline
void f(int* input, int size, int* output) {
    for (int i = 0; i < size; i++)
        output[i] = input[i];
}
```

- The compiler cannot *unroll* the loop (sequential execution, no ILP) because `output` and `input` pointers can be **aliased**, e.g. `output = input + 1`
- The aliasing problem is even worse for more complex code and *inhibits all kinds of optimization* including code re-ordering, vectorization, common sub-expression elimination, etc.

Most compilers (included `GCC/Clang/MSVC`) provide **restricted pointers** (`__restrict`) so that the programmer asserts that the pointers are not aliased

```
void f(int* __restrict input,
       int             size,
       int* __restrict output) {
    for (int i = 0; i < size; i++)
        output[i] = input[i];
}
```

Potential benefits:

- Instruction-level parallelism
- Less instructions executed
- Merge common sub-expressions

Benchmarking matrix multiplication

```
void matrix_mul_v1(const int* A,
                   const int* B,
                   int        N,
                   int*       C) {
```

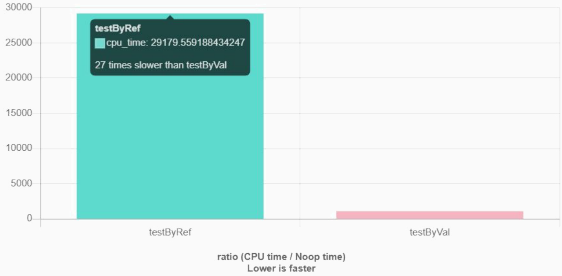
```
void matrix_mul_v2(const int* __restrict A,
                   const int* __restrict B,
                   int                   N,
                   int*       __restrict C) {
```

Optimization	-O1	-O2	-O3
v1	1,030 ms	777 ms	777 ms
v2	513 ms	510 ms	761 ms
Speedup	2.0x	1.5x	1.02x

```cpp
void foo(std::vector<double>& v, const double& coeff) {
    for (auto& item : v) item *= std::sinh(coeff);
}
```

vs.

```cpp
void foo(std::vector<double>& v, double coeff) {
    for (auto& item : v) item *= std::sinh(coeff);
}
```



C++ Objects

Variable/Object Scope

Declare local variable in the inner most scope

- the compiler can more likely fit them into registers instead stack
- it improves readability

Wrong:

```
int i, x;
for (i = 0; i < N; i++) {
    x     = value * 5;
    sum += x;
}
```

Correct:

```
for (int i = 0; i < N; i++) {
    int x   = value * 5;
    sum    += x;
}
```

Variable/Object Scope

Exception! Built-in type variables and passive structures should be placed in the inner most loop, while objects with constructors should be placed outside loops

```cpp
for (int i = 0; i < N; i++) {
    std::string str("prefix_");
    std::cout << str + value[i];
} // str call CTOR/DTOR N times
```

```cpp
std::string str("prefix_");
for (int i = 0; i < N; i++) {
    std::cout << str + value[i];
}
```

Object RAII Optimizations

- Prefer **direct initialization** and *full object constructor* instead of two-step initialization (also for variables)
- Prefer **move semantic** instead of copy constructor. Mark copy constructor as `=delete` (sometimes it is hard to see, e.g. implicit)
- Ensure defaulted default and copy constructors `= default` to enable vectorization

Object Dynamic Behavior Optimizations

- Avoid dynamic operations: **exceptions*** (and use `noexcept`), **`dynamic_cast`**, **smart pointer**

- **Virtual calls** are slower than standard functions

- Mark `final` all `virtual` functions that are not overridden

*Investigating the Performance Overhead of C++ Exceptions

Object Operation Optimizations

- Use `static` for all members that do not use instance member (avoid passing `this` pointer)

- Avoid multiple `+` operations between objects to avoid temporary storage

- Prefer `++obj` / `--obj` (return `&obj`), instead of `obj++`, `obj--` (return old `obj`)

- Prefer `x += obj`, instead of `x = x + obj` → avoid the object copy

Object Implicit Conversion

```
struct A {   // big object
    int array[10000];
};
struct B {
    int array[10000];

    B() = default;

    B(const A& a) { // user-defined constructor
        std::copy(a.array, a.array + 10000, array);
    }
};
//--------------------------------------------------------------------
void f(const B& b) {}

A a;
B b;
f(b); // no cost
f(a); // very costly!! implicit conversion
```