Modern C++ Programming

24. Performance Optimization II

Code Optimization

Federico Busato

2026-01-06

Table of Contents

1. **I/O Operations**
   - `printf`
   - Memory Mapped I/O
   - Speed Up Raw Data Loading
2. **Memory Optimizations**
   - Heap Memory
   - Stack Memory
   - `constexpr` vs. `static constexpr`
   - Cache Utilization
   - Memory Alignment
   - Memory Prefetch

Table of Contents

3 Arithmetic Types

- Data Types
- Arithmetic Operations
- Conversion
- Floating-Point
- Compiler Intrinsic Functions
- Value in a Range
- Lookup Table

4 Control Flow

- Branches
- Branch Hints - `[[likely]] / [[unlikely]]`
- Signed/Unsigned Integers
- Loops
- Loop Hoisting
- Loop Unrolling
- Assertions
- Compiler Hints - `[[assume]]/std::unreachable()`
- Recursion

Table of Contents

5 Functions

- Function Call Cost
- Argument Passing
- Function Inlining
- Pure Functions
- Constant Functions
- Pointers Aliasing

6 Object-Oriented Programming

7 `Std` Library and Other Language Aspects

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 <u>strictly</u> 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


```cpp
#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


```cpp
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;
    }
}
```

```
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 → **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`

Speed Up Raw Data Loading 2/2

Performance comparison of different methods for a file of 4.8 GB of integers. They are explicit values in a text file in the case of `ifstream` and `memory mapped`, while binary values for LZ4

Load Method	Exec. Time	Speedup
`ifstream + parsing`	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 interact with the operating system
- *Many small heap allocations are more expensive than one large memory allocation*
  The default page size on Linux is 4 KB. For smaller/multiple sizes, C++ uses a sub-allocator
- *Allocations within the page size is faster than larger allocations* (sub-allocator)

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 because it avoids filling the stack each time the function is reached.

`constexpr` vs. `static constexpr`

`constexpr` and `static constexpr` variables could produce very different code.

- Stack-local `constexpr` variable can be slightly faster than `static constexpr` for sizes less than 136–144 bytes .
- Larger data, such as greater than ∼2KB, copying into the stack becomes very expensive, making `static constexpr` much faster.
- `static constexpr` is faster with lower optimization options (`-O0` , `-O1`).
- `clang` and `msvc` are generally emits identical code in both cases.
- `constexpr` variable with dynamic indexing produces very inefficient code with GCC.

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

C++ Weekly - Ep 315 - constexpr vs static constexpr

Cache Utilization

Maximize cache utilization:

- Maximize spatial and temporal locality (see next examples)
- Prefer small data types
- For basic set query and insertion:
  - Prefer `std::vector<bool>` over a *dynamic* array of `bool`
  - Prefer `std::bitset` over `std::vector<bool>` if the data size is known in advance or bounded. *Fixed-size* array of `bool` should be always replaced by `std::bitset`
  - Remember that common std algorithms could not be optimized for these containers, e.g. `std::count_if`, `std::find`
- Prefer *stack* data structures *instead* of heap data structures, e.g. `std::vector` vs. `static_vector`

`A`, `B`, `C` matrices of size $N \times N$

`C = A * B`

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        int sum = 0;
        for (int k = 0; k < N; k++)
            sum += A[i][k] * B[k][j]; // row × column
        C[i][j] = sum;
    }
}
```

`C = A * B`^{T}

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        int sum = 0;
        for (int k = 0; k < N; k++)
            sum += A[i][k] * B[j][k]; // row × row
        C[i][j] = sum;
    }
}
```

Benchmark:

N	64	128	256	512	1024
`A * B`	$<$ 1 ms	5 ms	29 ms	141 ms	1,030 ms
`A * B`^{T}	$<$ 1 ms	2 ms	6 ms	48 ms	385 ms
`Speedup`	/	2.5x	4.8x	2.9x	2.7x

Temporal-Locality Example

Speeding up a random-access function

```
for (int i = 0; i < N; i++)         // V1
    out_array[i] = in_array[hash(i)];
```

```
for (int K = 0; K < N; K += CACHE) { // V2
    for (int i = 0; i < N; i++) {
        auto x = hash(i);
        if (x >= K && x < K + CACHE)
            out_array[i] = in_array[x];
    }
}
```

`V1`: 436 ms, `V2`: 336 ms → 1.3x speedup (temporal locality improvement)

.. but it needs a careful evaluation of `CACHE`, and it can even decrease the performance for other sizes

pre-sorted `hash(i)`: 135 ms → 3.2x speedup (spatial locality improvement)

lemire.me/blog/2019/04/27

Memory Alignment

Memory alignment refers to placing data in memory at addresses that conform to certain boundaries, typically powers of two (e.g., 1, 2, 4, 8, 16 bytes, etc.)

Note: For multidimensional data, alignment only means that the start address of the data is aligned, not that all start offsets for all dimensions are aligned., e.g. for a 2D matrix, if `row[0][0]` is aligned doesn't imply that `row[1][0]` has the same property. Also the strides between rows need to be multiple of the alignment

Data alignment is classified in:

- **Internal alignment** for struct/class layout optimization → reducing memory footprint, optimizing memory bandwidth, and minimizing cache-line misses
- **External alignment** across several elements of the same type → minimizing cache-line misses, vectorization (SIMD instructions)

Internal Structure Alignment

```
struct A1 {
   char   x1; // offset 0
   double y1; // offset 8!! (not 1)
   char   x2; // offset 16
   double y2; // offset 24
   char   x3; // offset 32
   double y3; // offset 40
   char   x4; // offset 48
   double y4; // offset 56
   char   x5; // offset 64 (65 bytes)
}
```

```
struct A2 {   // internal alignment
   char   x1; // offset 0
   char   x2; // offset 1
   char   x3; // offset 2
   char   x4; // offset 3
   char   x5; // offset 4
   double y1; // offset 8
   double y2; // offset 16
   double y3; // offset 24
   double y4; // offset 32 (40 bytes)
}
```

(1) We are wasting 40% of memory for (`A1`)

(2) Considering an *array of structures* (*AoS*) and a cache line of 64 bytes (x64 processors), every access to `A1` involves two cache line operations ($\sim$**2x slower**)

External Structure Alignment

In addiction to internal layout problems, even the structure `A2` introduces overhead if organized in an array. Loads lead to one or two cache line operations depending on the alignment at a specific index, e.g.

`index 0` $\rightarrow$ one cache line load

`index 1` $\rightarrow$ two cache line loads

It is possible to fix the structure alignment in two ways:

- **Memory padding** refers to manually introducing extra bytes at the end of the data structure to enforce memory alignment.
  e.g. add a `char` array of size 24 to the structure `A2`
- **Align keyword or attribute** allows specifying the alignment requirement of a type or an object (next slide)

External Structure Alignment

- *Explicit* alignment/padding for **variable / struct declaration** → affects `sizeof(T)`

  C++11 : `alignas(N)`

  GCC/Clang : `__attribute__((aligned(N)))`

  MSVC : `__declspec(align(N))`

- *Explicit* alignment for **pointers**

  C++20 : `std::assume_aligned<N>(ptr)` (`<memory>`)

  C++17 : aligned `new` or `std::aligned_alloc(align, size)`

  GCC/Clang : `__builtin_assume_aligned(ptr, N)`

```
struct alignas(16) S1 { // C++11
    int x, y;
};
struct __attribute__((aligned(16))) S2 { // compiler-specific attribute
    int x, y;
};
constexpr auto DefaultAlilgn = __STDCPP_DEFAULT_NEW_ALIGNMENT__;

S1 s;                    // 16B alignment
alignas(16) int var[3]; // 16B alignment
auto ptr1 = new S1[10]; // Warning! no aligment guarantee

auto ptr2 = new int[100];                    // alignment: max(4B, DefaultAlilgn)
auto ptr3 = std::aligned_alloc(8, 4);        // C++17, alignment: max(8B, DefaultAlilgn)
auto ptr4 = __builtin_assume_aligned(ptr2, 16); // compiler-specific attribute
auto ptr5 = std::assume_aligned<16>(ptr2);   // C++20

auto ptr = new (sizeof(int), std::align_val_t{8}); // C++17, max(8B, DefaultAlilgn)
::operator delete (ptr, std::align_val_t{8});
```

Memory Prefetch

`__builtin_prefetch` is used to *minimize cache-miss latency* by moving data into a cache before it is accessed. It can be used not only for improving *spatial locality*, but also *temporal locality*

```
for (int i = 0; i < size; i++) {
    auto data = array[i];
    __builtin_prefetch(array + i + 1, 0, 1); // 2nd argument, '0' means read-only
                                             // 3th argument, '1' means
                                             // temporal locality=1, default=3
    // do some computation on 'data', e.g. CRC
}
```

Alternatively, `-fprefetch-loop-arrays` can be used to emit prefetching instructions

The pros and cons of explicit software prefetching

Multi-Threading and Caches

The **CPU/threads affinity** controls how a process is mapped and executed over multiple cores (including sockets). It affects the process performance due to core-to-core communication and cache line invalidation overhead

Maximizing threads *"clustering"* on a single core can potentially lead to higher cache hits rate and faster communication. On the other hand, if the threads work independently/almost independently, namely they show high locality on their working set, mapping them to different cores can improve the performance

Arithmetic Types

Hardware Notes

- Instruction throughput greatly depends on processor model and characteristics, e.g., there is no hardware support for integer division on GPUs. This operation is translated to 100 instructions for 64-bit operands

- 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

- Arithmetic **increment/decrement** `x++` / `x--` has the same performance of `x += 1` / `x -= 1`
- Arithmetic **compound operators** (`a *= b`) has the same performance of assignment + operation (`a = a * b`) *
- **Prefer prefix increment/decrement** (`++var`) instead of the postfix operator (`var++`) *

* the compiler automatically applies such optimization whenever possible. This is not ensured for object types

- **Keep near constant values/variables** → the compiler can merge their values. Floating-point values requires more attention due to non-associativity
- Some operations on **`unsigned types`** are faster than on **`signed types`** because they don't have to deal with negative numbers, e.g. `x / 2` → `x » 1`
- Some operations on **`signed types`** are faster than on **`unsigned types`** because they can exploit *undefined behavior*, see next slide
- Prefer **logic operations** `||` to **bitwise operations** `|` to take advantage of short-circuiting

```
bool mainGuT(uint32_t i1, uint32_t i2, // if i1, i2 are int32_t, the code
             uint8_t *block) {         // uses half of the instructions!!
    uint8_t c1, c2;
    // 1                               // why? if i1, i2 are uint32_t the compiler
    c1 = block[i1], c2 = block[i2];    // must copy them into 32-bit registers to
    if (c1 != c2) return (c1 > c2);    // ensure wrap-around behavior before passing
    i1++, i2++;                        // them to the subscript operator (size_t)

    // 2                               // On the other hand, int32_t overflow is
    c1 = block[i1], c2 = block[i2];    // undefined behavior and the compiler can
    if (c1 != c2) return (c1 > c2);    // assume it never happens
    i1++, i2++;
    // ... continue repeating the      // the code is also optimal with size_t on 64-bit
}   //     code multiple times         // arch because block cannot be larger than it
```

Garbage In, Garbage Out: Arguing about Undefined Behavior with Nasal Daemons, Chandler Carruth, CppCon 2016

Arithmetic Operations - Integer Multiplication

Integer multiplication requires double the number of bits of the operands

```
// 32-bit platforms

int f1(int x, int y) {
    return x * y;                       // efficient, everything is 32-bit
}                                       // can overflow

int64_t f2(int64_t x, int64_t y) {      // not efficient, the compiler emulated
    return x * y;                       // 64-bit operations with 32-bit
}                                       // instructions
                                        // same for f2(int x, int64_t y)
int64_t f3(int x, int y) {
    return x * static_cast<int64_t>(y); // efficient!! the compiler knows that
}                                       // the inputs are 32-bit and the
                                        // multiplication requires 64-bit,
                                        // so no emulation is needed
```

Arithmetic Operations - Power-of-Two Multiplication/Division/Modulo

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

- 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

- 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 precomputed 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];
}
```

Make your lookup table do more

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

Computation is faster than decision

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`)

- Prefer `switch` statements to 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`

- In general, a *branch* has negligible effect on performance if it is not taken

- 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`

- Branch predictor: How many 'if's are too many?
- Is this a branch?

Hardware Features to Mitigate 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

Branch Hints - `[[likely]]` / `[[unlikely]]`

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;
}
```

Signed/Unsigned Integers

- Prefer **signed integer** for **loop indexing**. The compiler optimizes more aggressively such loops because integer overflow is not defined. Unsigned loop indexing generates complex intermediate expressions, especially for nested loops, that the compiler could not solve

- Prefer **32-bit signed integer** or **64-bit integer** for **any operation that is translated to 64-bit**. The most common is *array indexing*. The subscript operator implicitly defines its parameter as `size_t`. Any indexing operation with 32-bit unsigned integer requires the compiler to enforce wrap-around behavior, e.g. by moving the variable to a 32-bit register

```
unsigned v = ...;
// some operations on v
array[v];
```

Loops

- Prefer **square brackets** syntax `[]` over pointer arithmetic operations for array access to facilitate compiler loop optimizations (e.g. polyhedral loop transformations)

- *Range-based* loop could provide minor performance improvements for small loops that iterate over a container [1]

- On the other hand, *range-based loops* and *iterators* could inhibit many optimizations such as loop unrolling and vectorization

[1] The Little Things: Everyday efficiencies

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 can make your code better/faster:

+ Improve instruction-level parallelism (ILP)
+ Allow vector (SIMD) instructions
+ Reduce control instructions and branches

Loop unrolling can make your code worse/slower:

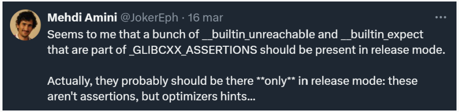
- Increase compile-time/binary size
- Require more instruction decoding
- Use more memory and instruction cache

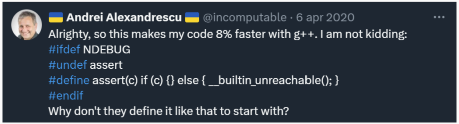
Unroll directive The `Intel`, `IBM`, `Arm`, `Nvidia`, `clang`, and GCC compilers provide the preprocessing directive `#pragma unroll` (`#pragma GCC unroll` for GCC) to insert above the loop to force loop unrolling. The compiler already applies the optimization in most cases

`Why are unrolled loops faster?`

Assertions

Some compilers (e.g. `clang`) use assertions for optimization purposes: most likely code path, not possible values, etc. [3]





Compiler Hints - `[[assume]]/std::unreachable()`

C++23 allows defining an *assumption* in the code that is always true

```
int x = ...;
[[assume(x > 0)]]; // the compiler assume that 'x' is positive

int y = x / 2;     // the operation is translated in a single shift as for
                   // the unsigned case
```

C++23 also provides `std::unreachable()` (`<utility>`) for marking unreachable code

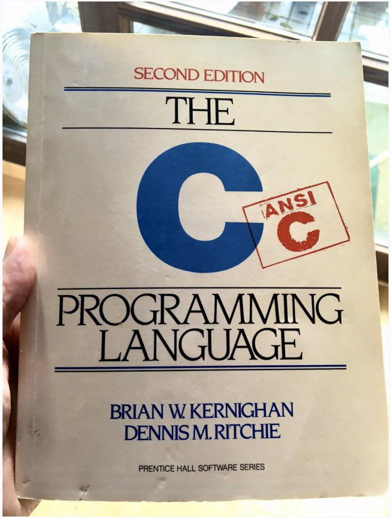
Compilers provide non-portable instructions for previous C++ standards: `__builtin_assume()` (clang), `__builtin_unreachable()` (gcc), `__assume()` (msvc)

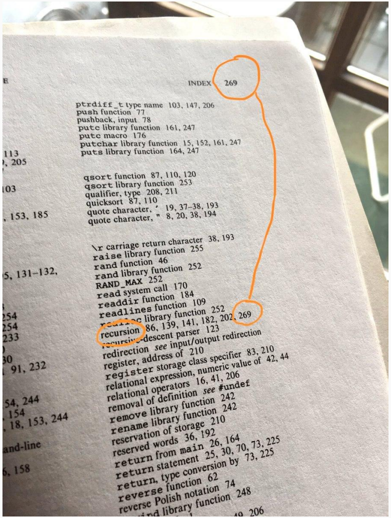
Note: sometimes user-provided information leads to worse optimization, see `@llvm.assume blocks optimization` and `Refined Input, Degraded Output: The Counterintuitive World of Compiler Behavior`

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

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 maintaining 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
recursion 86, 139, 141, 182, 202, 269
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

Functions

Function Call Cost

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 (same translation unit or Link-time-optimization)

Direct/Indirect 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

The **optimal way** to pass and return arguments (*by-value*) to/from functions is in *registers*. It also avoid the pointer aliasing performance issue. The following conditions must be satisfied:

- The object is **trivially copyable**: No user-provided copy/move/default constructors, destructor, and copy/move assignment operators, no virtual functions, apply recursively to base classes and non-static data members
- Linux/Unix (`SystemV x86-64 ABI`): data types $\leq$ **16 bytes** (8B $\times$ 2), max **6 arguments**
- Windows (`x64 ABI`): data types $\leq$ **8 bytes**, max **4 arguments**

- `when are structs/classes passed and returned in registers?`
- `System V ABI - X86-64 Calling Convention`
- `x64 calling convention - Parameter Passing`

- If the previous conditions are not satisfied, the object is passed **by-reference**. In addition, objects that are not *trivially-copyable* could be expensive to pass *by-value* (copied).

- Pass **by-reference** and **by-pointer** introduce one level of indirection

- Pass **by-reference** is more efficient than pass **by-pointer** because it facilitates variable elimination by the compiler, and the function code does not require checking for `NULL` pointer

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

According to Rome, "A seasoned performance engineer was looking through Strobelight data and discovered that by filtering on a particular `std::vector` function call (using the symbolized file and line number) he could identify computationally expensive array copies that happen unintentionally with the '`auto`' keyword in C++."

*After finding one of these costly array copies in the path of one of **Meta**'s major ad services, the engineer determined that the vector copy wasn't intentional. So he added an "`&`" after the `auto` keyword to turn the copy into a reference, which avoids unnecessary data duplication by pointing to the data rather than reproducing it.*

*"It was a **one-character commit**, which, after it was shipped to production, equated to an estimated **15,000 servers in capacity savings per year**," said Rome.*

`Meta recently made a 1 character change to their codebase which saves the equivalent of 15,000 servers in capacity per year` `const` modifier applied to values, pointers, references *does not produce better code* in most cases, but it is useful for ensuring read-only accesses

In some cases, pass `by-const` is beneficial for performance because `const` member function overloading could be cheaper than their counterparts

`GoTW#81: Constant Optimization?`

`inline` Function Declaration

inline

`inline` specifier for optimization purposes is just a hint for the compiler that increases the heuristic threshold for **inlining**, namely copying the function body where it is called

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

- the compiler can ignore the hint
- *Inlining* can be very effective for performance because it can merge different functions into one, allowing constant propagation, eliminating dead code, or combining instructions
- *inlined* 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).
- `-Winline` warns when a function marked inline could not be substituted, and gives the reason for the failure.

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

`inline` Function Declaration

Compiler extensions allow to manually *force* inline/non-inline functions with attributes. Here C++17attributes are adopted:

GCC/Clang:

```
[[gnu::always_inline]] void f() { ... }
[[gnu::noinline]]      void f() { ... }
```

MSVC:

```
[[msvc::forceinline]]  void f() { ... }
[[msvc::noinline]]     void f() { ... }
```

Inlining and Linkage

- A function with **internal linkage** is not visible outside the current translation unit, allowing the compiler to perform aggressive *inlining*.
- On the other hand, **external linkage** does not prevent function inlining if the function body is visible in a translation unit. However, function side effects or internal state may influence the compiler's decision.
- Functions with **external linkage** that are defined in a different translation unit cannot be inlined. *Link-time optimization* (LTO) may enable function inlining across translation units.

Symbol Visibility

All compilers, except MSVC, export all function symbols → the symbols can be used in other translation units and may influence the compiler's inlining decision.

Alternatives:

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

gcc.gnu.org/wiki/Visibility

Pure functions have *no side effect* on its parameters and don't modify global variables.
Pure functions can access global variables. On the other hand, they can only call other *pure functions*, cannot modify input pointers or references, and must have non- `void` return type.

Main property: `Referential transparency` → *Pure functions* always returns the same output for the same inputs, without affecting the program state.

Pure functions allow the following optimizations:

- *Common Sub-expression Elimination*: when the function is called multiple times with the same arguments.
- *Dead-Code Elimination*: if the result value or subsequent operations are not used, the function can be removed because it doesn't modify the program state.

```cpp
double pow2(double x); // defined in another translation unit

double caller(double v) {
    double x = pow2(v) + pow2(v); // generates 4 calls
    double y = pow2(v) + pow2(v);
    return x + y;
}
```

The compiler automatically recognizes *pure functions* when they are entirely visible in a translation unit. *Link-Time-Optimization* can help to optimize them across translation units. ASM statements prevent their detection.

Clang/GCC allow to explicitly marked *pure function* with the attribute `[[gnu::pure]]`.

Constant Functions

Constant Functions have *no side effect* on its parameters and don't refer global variables. They are a stricter case of *pure functions*.

Constant functions allow further optimizations:

- *Strong Common Sub-expression Elimination*: this optimization can be applied even for non-subsequent statements because modification of the global state don't affect them.
- *Loop optimizations*: thanks to the above property, *constant functions* within a loop can be reordered arbitrarily

Clang/GCC allow to explicitly marked *constant function* with the attribute `[[gnu::const]]`.

Implications of pure and constant functions

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

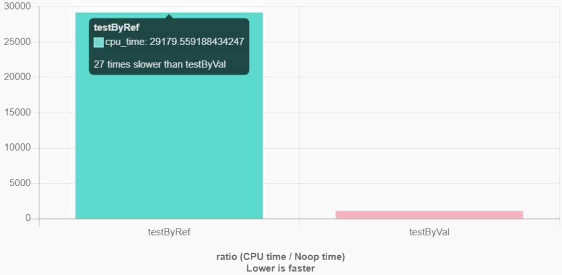
```
void matrix_mul_v1(const int* A,
                   const int* B,
                   int        N,
void matrix_mul_v2(const int* __restrict A,
                   const int* __restrict B,
                   int                   N,
                   int*       __restrict C) {
```

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

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

vs.

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



Object-Oriented Programming

Declare local variable in the innermost scope

- the compiler can more likely fit them into registers instead of 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;
}
```

- C++17 allows local variable initialization in `if` and `switch` statements, while C++20 introduces them for in *range-based loops*

Variable/Object Scope

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

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

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

Object 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)

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

- If the object semantic is *trivially copyable*, ensure **defaulted** `= default` *default/copy constructors* and *assignment operators* to enable vectorization

Object Dynamic Behavior Optimizations

- **Virtual calls** are slower than standard functions
  - Virtual calls prevent any kind of optimizations as function lookup is at runtime (loop transformation, vectorization, etc.)
  - Virtual call overhead is up to 20%-50% for function that can be inlined

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

- Avoid dynamic operations, e.g. `dynamic_cast`

- The Hidden Performance Price of Virtual Functions
- Investigating the Performance Overhead of C++ Exceptions

Object Operation Optimizations

- Minimize multiple `+` operations between objects to avoid temporary storage
- Prefer `x += obj`, instead of `x = x + obj` → avoid object copy and temporary storage
- Prefer `++obj` / `-obj` (return `&obj`), instead of `obj++`, `obj-` (copy and return old `obj`)

Object Implicit Conversion

```cpp
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
```

`Std` Library and Other Language Aspects

- Avoid old `C` library routines such as `qsort`, `bsearch`, etc. Prefer **`std::sort`**, **`std::binary_search`** instead
  - `std::sort` is based on a hybrid sorting algorithm. Quick-sort / head-sort (introsort), merge-sort / insertion, etc. depending on the `std` implementation
  - Prefer `std::find()` for small array, `std::lower_bound`, `std::upper_bound`, `std::binary_search` for large sorted array

Function Optimizations

- `std::fill` applies `memset` and `std::copy` applies `memcpy` if the input/output are continuous in memory

- Use the same type for initialization in functions like `std::accumulate()`, `std::fill`

```
auto array = new int[size];
...
auto sum = std::accumulate(array, array + size, 0u);
// 0u != 0 → conversion at each step

std::fill(array, array + size, 0u);
// it is not translated into memset
```

Containers

- Use `std` container member functions (e.g. `obj.find()`) instead of external ones (e.g. `std::find()`). Example: `std::set` $O(log(n))$ vs. $O(n)$
- Be aware of container properties, e.g. `vector.push_back(v)`, instead of `vector.insert(vector.begin(), value)` $\rightarrow$ entire copy of all vector elements
- Set `std::vector` size during the object construction (or use the `reserve()` method) if the number of elements to insert is known in advance $\rightarrow$ every implicit resize is equivalent to a copy of all vector elements
- Consider *unordered* containers instead of the standard one, e.g. `unordered_map` vs. `map`
- Prefer `std::array` instead of dynamic heap allocation

Critics to Standard Template Library (STL)

- Platform/Compiler-dependent implementation
- Execution order and results across platforms
- Debugging is hard
- Complex interaction with custom memory allocators
- Error handling based on exceptions is non-transparent
- Binary bloat
- Compile time (see `C++ Compile Health Watchdog`, and `STL Explorer`)

`STL isn't for *anyone*`

Exceptions

Exceptions affect both performance and memory resources:

- Exceptions make functions harder to inline, due to their side effect.
- Side effects also prevent or limit common optimizations, such as instruction reordering and loop unrolling.
- Exceptions produce code bloat affecting code locality and decreasing cache hits.

Mitigation:

- Use `noexcept` decorator, especially for move constructor and assignment → program is aborted if an error occurs.
- Some benchmarks report 5-7% performance improvement `noexcept` Can (Sometimes) Help (or Hurt) Performance
- Bitcoin: 9% less memory: make `SaltedOutpointHasher` `noexcept`

Other Language Aspects

- Prefer `lambda` expression (or `function object`) instead of `std::function` or function pointers
- Avoid dynamic operations: **exceptions** (and use `noexcept`), **smart pointer** (e.g. `std::unique_ptr`)