

KIV/CPP – Programming in C++

12. Compilation of selected constructs into machine code, optimization

Martin Úbl

KIV ZČU

2025/2026

- C++ is a high-level language
- the connection to low-level code is not removed
 - it is only abstracted away
 - reference
 - STL containers and structures
 - and more...
- orders of magnitude higher demands on the compiler in order to produce “equally” efficient code as in pure C
- optimization

- language features must be translated into low-level routines (even without optimizations)
 - classes, methods, and attributes
 - polymorphism
 - references
 - templates
 - return values
 - ...
- high-level constructs can be optimized away completely
 - access to an array/vector element
 - STL algorithms
 - ...

- classes, methods, and attributes
- low-level code does not know classes (or objects)
- we must somehow express that a method belongs to an object
- a class instance (object) in low-level code “degenerates” into a structure instance
 - attributes become structure members
- methods are transformed into ordinary functions
- a pointer to the structure (originally the object) is typically passed as the “0th parameter”
 - `this`
 - or rather the `thiscall` calling convention

Abstraction

// C++

```
class Number {  
    private:  
        int number;  
    public:  
        void setNumber(int c);  
        int getNumber();  
}
```

// C

```
struct Number {  
    int number;  
}  
  
void setNumber(Number* this, int c);  
int getNumber(Number* this);
```

- virtual methods
- recap
- low-level code does not know polymorphism
- we must somehow express that a method belongs to a specific type
- virtual method table
 - array of pointers to functions
 - in an object as the “zeroth” attribute – a pointer to the table
 - e.g., MSVS inserts the symbol `__vfptr`

```
class Number {  
    protected:  
        int number;  
    public:  
        Number() : number{ 0 } {}  
        Number(int a) : number{ a } {}  
  
        virtual int get() {  
            return number;  
        }  
        virtual int getDoubled() {  
            return number * 2;  
        }  
};
```

- virtual methods
- method invocation by selecting an address from the vtable
- call

```
a->get();  
00007FF77985111D  mov     rax,qword ptr [rsi]  
00007FF779851120  mov     rcx,rsi  
00007FF779851123  call    qword ptr [rax]  
a->getDoubled();  
00007FF779851125  mov     rax,qword ptr [rsi]  
00007FF779851128  mov     rcx,rsi  
00007FF77985112B  call    qword ptr [rax+8]
```

Figure: The class has two virtual methods, in the order `get()` and `getDoubled()`

```
class NumberSquared : public Number {
public:
    NumberSquared() : Number{ } {}
    NumberSquared(int a) : Number{ a } {}

    virtual int get() override {
        return number * number;
    }
    virtual int getDoubled() override {
        return number * number * 2;
    }
};
```

Abstraction

- virtual methods
- method invocation by selecting an address from vtable
- call

```
    b->get();  
00007FF7D7D1111D  mov     rax,qword ptr [rsi]  
00007FF7D7D11120  mov     rcx,rsi  
00007FF7D7D11123  call    qword ptr [rax]  
    b->getDoubled();  
00007FF7D7D11125  mov     rax,qword ptr [rsi]  
00007FF7D7D11128  mov     rcx,rsi  
00007FF7D7D1112B  call    qword ptr [rax+8]
```

Figure: The class overrides both virtual methods of the parent

Abstraction

- virtual methods
- what if we explicitly instantiate a descendant?
- the compiler *could understand* that it does not have to use polymorphism
 - but this does not happen (not always)

```
NumberSquared* c = new NumberSquared(10);
```

```
c->get();  
00007FF6C34A111D  mov     rax,qword ptr [rbx]  
00007FF6C34A1120  mov     rcx,rbx  
00007FF6C34A1123  call    qword ptr [rax]
```

Figure: Explicitly typed descendant

Abstraction

- virtual methods
- what if we explicitly instantiate a descendant?
- we must help it with the `final` keyword (on the class or method)
- so-called *devirtualization* is performed

```
virtual int get() override final { ... }
```

```
    acc += c->get();  
00007FF71C9A1114  mov     rcx,rbx  
00007FF71C9A1117  call    CislonaDruhou::get (07FF71C9A1060h)  
00007FF71C9A111C  mov     ecx,dword ptr [acc]  
00007FF71C9A1120  add     eax,ecx
```

Figure: Explicitly typed descendant, with `final`

- return values
- from a function/method, we return a class instance constructed inside
 - initialization in the return value
 - initialization along the way and return of a named object
- *(Named) Return Value Optimization* (RVO, NRVO)
- to prevent multiple copies, the compiler deduces that this is unnecessary in the given context and constructs the object only once

```
std::vector<int> gen_fixed_vector(int a) {  
    return { a, a * 2, a * 3 };  
}
```

- NRVO

```
std::vector<int> gen_vector(size_t n)
{
    std::vector<int> vec(n);

    // ... generation ...

    return vec;
}

auto vec = gen_vector(10);
```

- accesses, e.g., into a vector and array, can be optimized
- although this is a method call, it is abstracted away thanks to inlining
- access should then be equivalent to access into a C-style array
 - `std::array` and a statically allocated array
 - `std::vector` and a dynamically allocated array

```
// MSVS implementation of vector operator []
_Ty& operator[](const size_type _Pos) {
    return (this->_Myfirst())[_Pos];
}
```

acc += vec[0]; acc += vec[1]; acc += vec[2]; acc += vec[4];			acc += vec[0]; acc += vec[1]; acc += vec[2]; acc += vec[4];		
00007FF72C711233	lea	rdx,[rdi+rbx]	00007FF6DB81122C	mov	rdx,qword ptr [rsp+40h]
00007FF72C711237	add	rdx,qword ptr [rsp+40h]	00007FF6DB811238	add	rdx,rdi
00007FF72C71123C	add	rdx,rsi	00007FF6DB81123B	add	rdx,rbx
			00007FF6DB81123E	add	rdx,rsi

Figure: `std::array` (left) vs. static array (right)

- note: array elements are cached in registers in both cases, so `rdi`, `rbx`, `rsi`, ... are added
- note 2: the array contents are on the stack, so `rsp` is also involved

acc += vec[0];				acc += vec[0];			
acc += vec[1];				acc += vec[1];			
acc += vec[2];				acc += vec[2];			
acc += vec[4];				acc += vec[4];			
00007FF73ACE15CB	mov	rbx,qword ptr [vec]		00007FF6A158165A	mov	rdx,qword ptr [rax+20h]	
00007FF73ACE15D0	mov	rdx,qword ptr [rbx+20h]		00007FF6A158165E	add	rdx,qword ptr [rax+10h]	
00007FF73ACE15D4	add	rdx,qword ptr [rbx+10h]		00007FF6A1581662	add	rdx,qword ptr [rax+8]	
00007FF73ACE15D8	add	rdx,qword ptr [rbx+8]		00007FF6A1581666	add	rdx,qword ptr [rax]	
00007FF73ACE15DC	add	rdx,qword ptr [rbx]					

Figure: `std::vector` (left) vs. dynamically allocated array (right)

- note: the contents of the vector/dyn. allocated array are on the heap – for the vector, the base address is first loaded into `rbx`; in the second case, for some reason, the base address is already in `rax`

Strength reduction

- the compiler tries to use faster operations to achieve an equivalent result
- e.g., multiplication is more demanding than addition, and therefore

```
for (int i = 0; i < 1000; i++)  
    func(i * 9999);
```

- compilation will most likely change this to

```
for (int i = 0; i < 9999000; i += 9999)  
    func(i);
```

Strength reduction

- a very common effort is to avoid division at all costs
- integer division is one of the most demanding operations in modern CPUs
- it can often be replaced by bit operations (and, or, bitshift, xor)
 - BUT... only under the assumption that the compiler knows the divisor in advance
 - in general, it is therefore better to divide by a compile-time constant
- e.g., division of unsigned long numbers on a 64-bit ALU (the compiler makes the change itself):

```
// approx. 96 cycles
```

```
y = x / 3;
```

```
// approx. 12 cycles
```

```
y = (x * 0xAAAAAAAAAB) >> 33;
```

Lookup table

- lookup tables are often used
- e.g., operations with a limited domain
- a few extra bytes of memory are often sacrificed for a table with precomputed results
- an example can be an alternative to the `popcnt` instruction on 1B
 - a processor instruction that counts set bits in the input
 - when we do not have this instruction, we must substitute for it
 - local variable, shifting, and manual counting of ones? too demanding (approx. 18 instructions)
 - we treat the value as an index into the table
 - the table contains precomputed counts of ones for all 1B numbers (256 values) (approx. 1-2 instructions)

Inlining

- a principle known outside C++ as well
- insertion of the function body at the place where it is called
- effective for small functions
 - removes the overhead of a function call
 - stack manipulation
 - parameter passing
 - return values
 - register overlap, ...
 - allows the inlined function to be optimized in the caller's context
 - this would not be possible without inlining because of code sharing across different contexts
- the `inline` keyword
- the compiler often deduces this itself, but it is good to mark functions this way

- *-fomit-frame-pointer*
- not using the frame pointer (BP/EBP/RBP register on x86)
- effective for small functions with a small number of local variables
- allows the frame pointer to be used for something else (a register instead of a memory reference)
- speeds up execution; access to a register is always faster
- however, it sometimes complicates life for some diagnostic tools
 - debugger
 - profiler

- *constant folding, constant propagation*
- the compiler detects which values do not change
- it may compute some of them at compile time (folding)
- this then allows better optimization
 - number of local variables (it puts the value directly into an instruction)
 - character of operations (e.g., integer multiplication by 2 is a bitshift)
 - and others...
- however, this is not always possible, therefore...
 - use `constexpr` and `constexpr`
 - if we know that something will be constant, we can replace it with a constant directly

- *omitting constant variable*
- even though we assign a constant to a variable and only then somewhere into memory, the compiler detects immutability and inserts the value directly into the instruction

```
int multiple = 5;  
*target = multiple;
```

```
mov    dword ptr [memory (07FF68FDD3628h)],5
```

- *operation substitution*
- e.g.
 - multiplication by two → addition to itself
 - multiplication by five → addition of the value to its quadruple (bitshift)

```
int multiple = argc * 5;  
variable = multiple * 2;
```

```
lea    eax,[rcx+rcx*4]  
add    eax,eax
```

- *operation substitution*
- multiplication by a power of two $\rightarrow$ left shift by the exponent

```
int multiple = argc * 5;  
memory = multiple * 1024;
```

```
lea    eax,[rcx+rcx*4]  
shl    eax,0Ah
```

Constants

- *operation substitution*
- but ...
- multiplication by a power of two stored in a variable → actual multiplication

```
int multiple = argc * 5;  
volatile int multiplier = 1024;  
memory = multiple * multiplier;
```

```
mov     dword ptr [rsp+8],400h  
lea     edx,[rcx+rcx*4]  
mov     eax,dword ptr [multiplier]  
imul    eax,edx
```

- *note: with the volatile keyword, we are substituting here for a “real context” in which the variable value is not known in advance*

Expression elimination

- *expression elimination*
- compiler detects expressions that can be evaluated only once
- e.g., an intermediate calculation in a loop that is invariant with respect to loop iterations
- such a calculation is performed only once and the result is stored in a local variable

```
int result = 0;
00007FF75DC01000 xor     r9d,r9d
00007FF75DC01003 mov     dword ptr [rsp+18h],64h
    for (size_t i = 0; i < pocetcyklu; i++) {
00007FF75DC01008 mov     eax,dword ptr [pocetcyklu]
00007FF75DC0100F mov     edx,r9d
00007FF75DC01012 imul    eax,ecx,29h ←
00007FF75DC01015 nop     word ptr [rax+rax]
00007FF75DC01020 movsxd  rcx,dword ptr [pocetcyklu] ←
    const int pocet = argc * 41;
    result += pocet;
00007FF75DC01025 add     r9d,eax
00007FF75DC01028 inc     rdx
00007FF75DC0102B cmp     rdx,rcx
00007FF75DC0102E jbe     main+20h (07FF75DC01020h)
    }
}
```

Figure: multiplication performed only once (green arrow), the loop repeats without it (red)

Recursion elision

- *recursion elision*
- under certain circumstances, recursion can be transformed into a loop
- e.g., so-called *tail recursion*
 - the function body ends with a recursive call to itself
- again, this can significantly save function-call overhead
- and, naturally, the stack as well

```
    return mul;

    arg--;
00007FF6F08C1005  leave
    mul += arg;
00007FF6F08C1006  add     edx,ecx
00007FF6F08C1008  test    ecx,ecx
00007FF6F08C100A  jne     do_recurse+4h (07FF6F08C1004h)
    return do_recurse(arg, mul);
}
00007FF6F08C100C  mov     eax,edx
00007FF6F08C100E  ret
```

Figure: function call replaced by a jump (together with other optimizations), effectively replaced by a loop

Loop Optimization

- loop optimization forms a large part of the optimization process
- with the advent of vector instructions, it becomes even more important
- they often require a known number of iterations
- e.g.
 - *loop unrolling* – multiplying the body of a for loop to reduce the number of condition checks
 - *loop reversal* – reversing the incrementing direction of a for loop to decrementing
 - *loop fission* – splitting one loop into multiple subloops because of cache locality
 - *loop fusion* – merging multiple loops with the same number of iterations into one (e.g., vectorization)
 - and more...

Loop Optimization

- *loop unrolling*
- reducing the number of condition checks, or vectorization
- original loop

```
for (size_t i = 0; i < 32; i++)  
    acc += vec[i];
```

- “unrolled” loop – done automatically by the compiler

```
for (size_t i = 0; i < 8; i++) {  
    acc += vec[i*4 + 0];  
    acc += vec[i*4 + 1];  
    acc += vec[i*4 + 2];  
    acc += vec[i*4 + 3];  
}
```

Loop unrolling

```
for (size_t i = 0; i < VectorSize; i++)  
    acc += vec[i];  
00007FF6469B1287 mov     rax,qword ptr [vec]  
00007FF6469B128C mov     rdx,qword ptr [rax+18h]  
00007FF6469B1290 add     rdx,qword ptr [rax+10h]  
00007FF6469B1294 add     rdx,qword ptr [rax+8]  
00007FF6469B1298 add     rdx,qword ptr [rax]
```

Figure: Loop unrolling, VectorSize = 4

```
    acc += vec[i];  
00007FF6C8BE12A0 vpaddq    ymm1,ymm1,ymmword ptr [rax+rbx*8]  
00007FF6C8BE12A5 vpaddq    ymm2,ymm2,ymmword ptr [rax+rbx*8+20h]  
    for (size_t i = 0; i < VectorSize; i++)  
00007FF6C8BE12AB add     rbx,8  
00007FF6C8BE12AF cmp     rbx,80h  
00007FF6C8BE12B6 jb      main+50h (07FF6C8BE12A0h)  
    int64_t acc = 0;  
00007FF6C8BE12B8 vpaddq    ymm2,ymm1,ymm2  
00007FF6C8BE12BC vextracti128 xmm1,ymm2,1  
00007FF6C8BE12C2 vpsrldq    xmm0,xmm1,8  
00007FF6C8BE12C7 vpaddq    xmm3,xmm1,xmm0  
00007FF6C8BE12CB vextracti128 xmm2,ymm2,0  
00007FF6C8BE12D1 vpsrldq    xmm0,xmm2,8  
00007FF6C8BE12D6 vpaddq    xmm0,xmm2,xmm0  
00007FF6C8BE12DA vpaddq    xmm1,xmm0,xmm3  
00007FF6C8BE12DE vmovq     rdx,xmm1
```

Figure: Loop unrolling, VectorSize = 128

- *loop reversal*
- reversing the direction of iteration
- original loop

```
for (size_t i = 0; i < VectorSize; i++)  
    acc += vec[i];
```

- reversed loop

```
for (size_t i = VectorSize - 1; i--; )  
    acc += vec[i];
```

- note: this uses the fact that `i--` returns 0 at the end of the loop, so the condition evaluates negatively and the loop ends

Loop reversal

```
for (size_t i = 0; i < vec.size(); i++)
00007FF7D9A01289 mov     eax,ebx
00007FF7D9A0128B mov     rcx,qword ptr [rsp+30h]
00007FF7D9A01290 mov     rdx,qword ptr [vec]
00007FF7D9A01295 sub     rcx,rdx
00007FF7D9A01298 sar     rcx,3
00007FF7D9A0129C test    rcx,rcx
00007FF7D9A0129F je      main+5Dh (07FF7D9A012ADh)
    acc += vec[i];
00007FF7D9A012A1 add     rbx,qword ptr [rdx+rax*8]
for (size_t i = 0; i < vec.size(); i++)
00007FF7D9A012A5 inc     rax
00007FF7D9A012A8 cmp     rax,rcx
00007FF7D9A012AB jnb     main+51h (07FF7D9A012A1h)

for (size_t i = vec.size()-1; i--; )
00007FF605AA128E mov     rcx,qword ptr [vec]
00007FF605AA1293 sub     rax,rcx
00007FF605AA1296 sar     rax,3
00007FF605AA129A sub     rax,1
00007FF605AA129E je      main+5Ch (07FF605AA12ACh)
00007FF605AA12A0 dec     rax
    acc += vec[i];
00007FF605AA12A3 add     rbx,qword ptr [rcx+rax*8]
for (size_t i = vec.size()-1; i--; )
00007FF605AA12A7 test    rax,rax
00007FF605AA12AA jne     main+50h (07FF605AA12A0h)
```

Figure: Loop reversal with unknown size; original loop on the left, reversed on the right

Loop reversal

```
for (size_t i = VectorSize - 1; i--; )
    acc += vec[i];
00007FF7A6231342 add     rdx,qword ptr [rax+320h]
00007FF7A6231349 add     rdx,qword ptr [rax+318h]
00007FF7A6231350 add     rdx,qword ptr [rax+310h]
00007FF7A6231357 add     rdx,qword ptr [rax+308h]
00007FF7A623135E add     rdx,qword ptr [rax+300h]
00007FF7A6231365 add     rdx,qword ptr [rax+2F8h]
00007FF7A623136C add     rdx,qword ptr [rax+2F0h]
00007FF7A6231373 add     rdx,qword ptr [rax+2E8h]
00007FF7A623137A add     rdx,qword ptr [rax+2E0h]
00007FF7A6231381 add     rdx,qword ptr [rax+2D8h]
00007FF7A6231388 add     rdx,qword ptr [rax+2D0h]
00007FF7A623138F add     rdx,qword ptr [rax+2C8h]
00007FF7A6231396 add     rdx,qword ptr [rax+2C0h]
00007FF7A623139D add     rdx,qword ptr [rax+2B8h]
00007FF7A62313A4 add     rdx,qword ptr [rax+2B0h]
00007FF7A62313AB add     rdx,qword ptr [rax+2A8h]
00007FF7A62313B2 add     rdx,qword ptr [rax+2A0h]
00007FF7A62313B9 add     rdx,qword ptr [rax+298h]
00007FF7A62313C0 add     rdx,qword ptr [rax+290h]
00007FF7A62313C7 add     rdx,qword ptr [rax+288h]
00007FF7A62313CE add     rdx,qword ptr [rax+280h]
00007FF7A62313D5 add     rdx,qword ptr [rax+278h]
00007FF7A62313DC add     rdx,qword ptr [rax+270h]
00007FF7A62313E3 add     rdx,qword ptr [rax+268h]
00007FF7A62313EA add     rdx,qword ptr [rax+260h]
00007FF7A62313F1 add     rdx,qword ptr [rax+258h]
00007FF7A62313F8 add     rdx,qword ptr [rax+250h]
00007FF7A62313FF add     rdx,qword ptr [rax+248h]
00007FF7A6231406 add     rdx,qword ptr [rax+240h]
00007FF7A623140D add     rdx,qword ptr [rax+238h]
00007FF7A6231414 add     rdx,qword ptr [rax+230h]
```

Figure: Loop reversal with known size, VectorSize = 128

Loop Optimization

- *loop fusion*
- merging multiple loops into one when they have the same number of iterations
- original loops

```
for (size_t i = 0; i < VectorSize; i++)  
    acc1 += vec[i];  
for (size_t i = 0; i < VectorSize; i++)  
    acc2 += vec[i]*2;
```

- merged loop

```
for (size_t i = 0; i < VectorSize; i++) {  
    acc1 += vec[i];  
    acc2 += vec[i]*2;  
}
```

- the “opposite” is loop fission (splitting)

- to conclude the purely language-oriented part – notes on writing efficient code
- partly a recap
- using efficient structures
- “helping” the compiler with code optimization

- static arrays
- `std::array`
- access to elements
 - the `[]` operator
 - use the `.at()` method only when you are not sure whether you are within bounds – it always checks bounds and may throw an exception, but it slows things down!

- dynamic arrays
- `std::vector`
- contiguous memory, inefficient deletion
- inserting elements
 - known count: `reserve(count) + push_back()`
 - for POD, `resize(count)` can also be used
 - unknown count: `push_back()`, `emplace_back()`
- access to elements
 - the `[]` operator
 - the `.at()` method – same implication as for array (slows things down)
 - iterator

- `list`
- `std::list`
- does not guarantee contiguous memory
- inserting elements
 - `push_back()`, `emplace_back()`
- access to elements
 - iterator
 - range-based `for`

- associative map
- `std::map`, `std::unordered_map`
- inserting elements
 - by index
 - `insert()`
- access to elements
 - iterator, `find()`
 - range-based `for`

- `std::map`
 - lower memory requirements
 - ordered elements
 - potentially slower
- `std::unordered_map`
 - higher memory requirements
 - unordered elements
 - potentially faster
- applies analogously to `std::set` and `std::unordered_set`

- polymorphism
- dynamic polymorphism is more expensive (vtable)
- avoid it if possible
 - static polymorphism (CRTP)
 - the `final` keyword on classes and methods
 - concepts to replace polymorphism in the context of generic programming
- overall, we try to limit the need to work with types at runtime
 - this is not always fully possible or appropriate

- preventing unnecessary copies
- sometimes the compiler does this itself (RVO/NRVO, copy elision)
- sometimes we must help it
 - move semantics
 - `emplace_back()` in containers
 - references

- allocation optimization
- dynamic allocation is more expensive
- we prefer static allocation if possible
 - small, unshared classes/POD
 - deterministic lifetime
- dynamic allocation should be wrapped in structures intended for that purpose
 - `unique_ptr` – for an unshared pointer
 - `shared_ptr` – for a shared pointer (note that it has higher overhead)
 - STL containers
- minimizing or completely eliminating the use of raw pointers

- use of exceptions
- Zero-Cost model
 - added overhead only if an exception occurs
- we use exceptions only for exceptional situations
 - exclusively error handling
- we should use classes derived from `std::exception`

Notes on Writing Efficient Code



- `const`
- evaluation at compile time
- `constexpr`, `constexpr`
 - constants
 - functions
- templates

- moderate use of templates
- too many template instances lead to huge code
- potentially less of it then fits into the CPU instruction cache
- find a balance between using templates and runtime resolution
- limit value parameterization

- the profiler is a friend
- when the code is still slower than it feels it should be
- but even if not – profiling should be part of the development process

And when everything else fails...

