Programming in C++

Why and How
Part 2

Andreas Füglistaler

Programming in C++

Why

- ✔ Old language, new features
- ✔ High performance code
- ✔ Reusable code

How

- ✔
- ✔
- ✔

Variables are unsafe by default

Variables are mutable
No buffer overflow check
No memory check

Make variables safe

Variables are mutable	➡	declare const/constexpr
No buffer overflow check	➡	use algorithm
No memory check	➡	use smart pointers

Functions are unsafe by default

Functions mutate references
Member-functions mutate members
Functions evaluate at run-time

Make variables safe

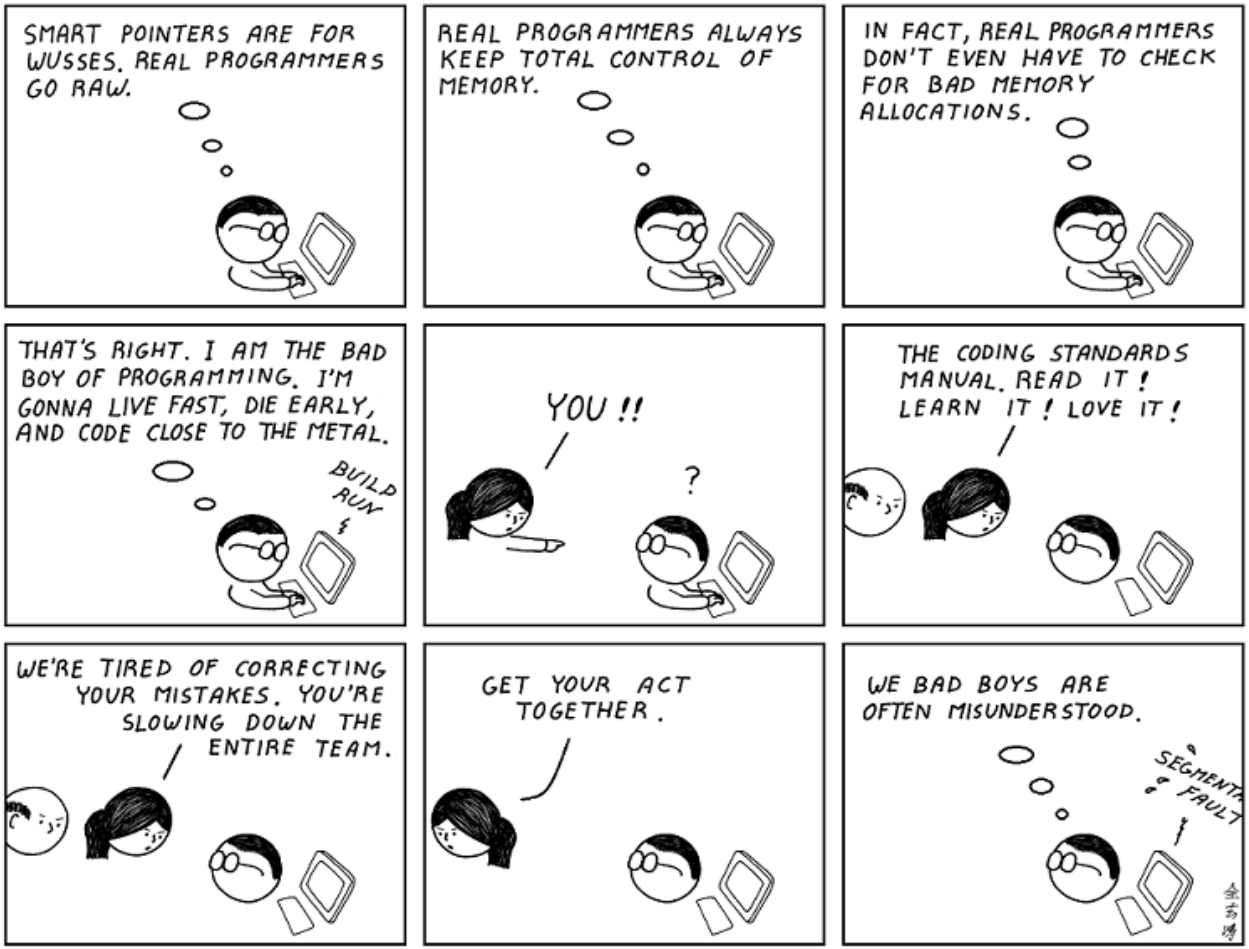
Functions mutate references	➡	const references
Member-functions mutate members	➡	const
Functions evaluate at run-time	➡	constexpr

Safe C++

- ✔ Declare variables as const
- ✔ Declare member-functions as const
- ✔ Declare functions as constexpr (if possible)

- ✔ Use algorithms, not raw loops
- ✔ Use objects, not pointers
- ✔ Use smart pointers, not raw pointers

- ✔ Use modern C++ features


SMART POINTERS ARE FOR WUSSES. REAL PROGRAMMERS GO RAW.
REAL PROGRAMMERS ALWAYS KEEP TOTAL CONTROL OF MEMORY.
IN FACT, REAL PROGRAMMERS DON'T EVEN HAVE TO CHECK FOR BAD MEMORY ALLOCATIONS.
THAT'S RIGHT. I AM THE BAD BOY OF PROGRAMMING. I'M GONNA LIVE FAST, DIE EARLY, AND CODE CLOSE TO THE METAL.
BUILD
RUN
YOU !!
?
THE CODING STANDARDS MANUAL. READ IT ! LEARN IT ! LOVE IT !
WE'RE TIRED OF CORRECTING YOUR MISTAKES. YOU'RE SLOWING DOWN THE ENTIRE TEAM.
GET YOUR ACT TOGETHER.
WE BAD BOYS ARE OFTEN MISUNDERSTOOD.
SEGMENT
FAULT

Premature Optimization

Premature optimization is the root of all evil.
(Donald Knuth, 1974)

Premature Optimization

Premature optimization is the root of all evil.
(Donald Knuth, 1974)

However

In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal; and I believe the same viewpoint should prevail in software engineering.

(Donald Knuth, **same article**)

No Micro-Optimization

x Do not do this:

```
std::array<int, 4> a;
a[0] = 0;
a[1] = 1;
a[2] = 2;
a[3] = 3;
```

✔ Do this:

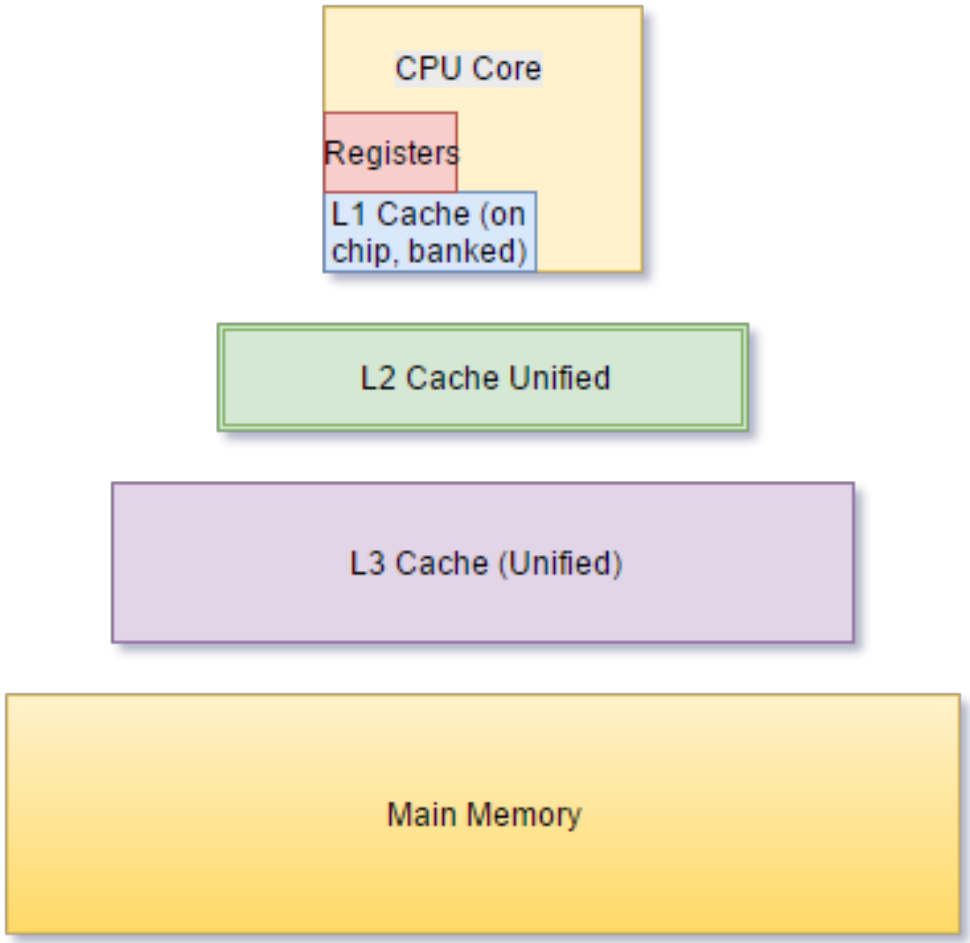
```
for (size_t i = 0; i < a.size(); ++i) a[i] = i;
```

✔ Even better:

```
std::iota(a.begin(), a.end(), 0));
```

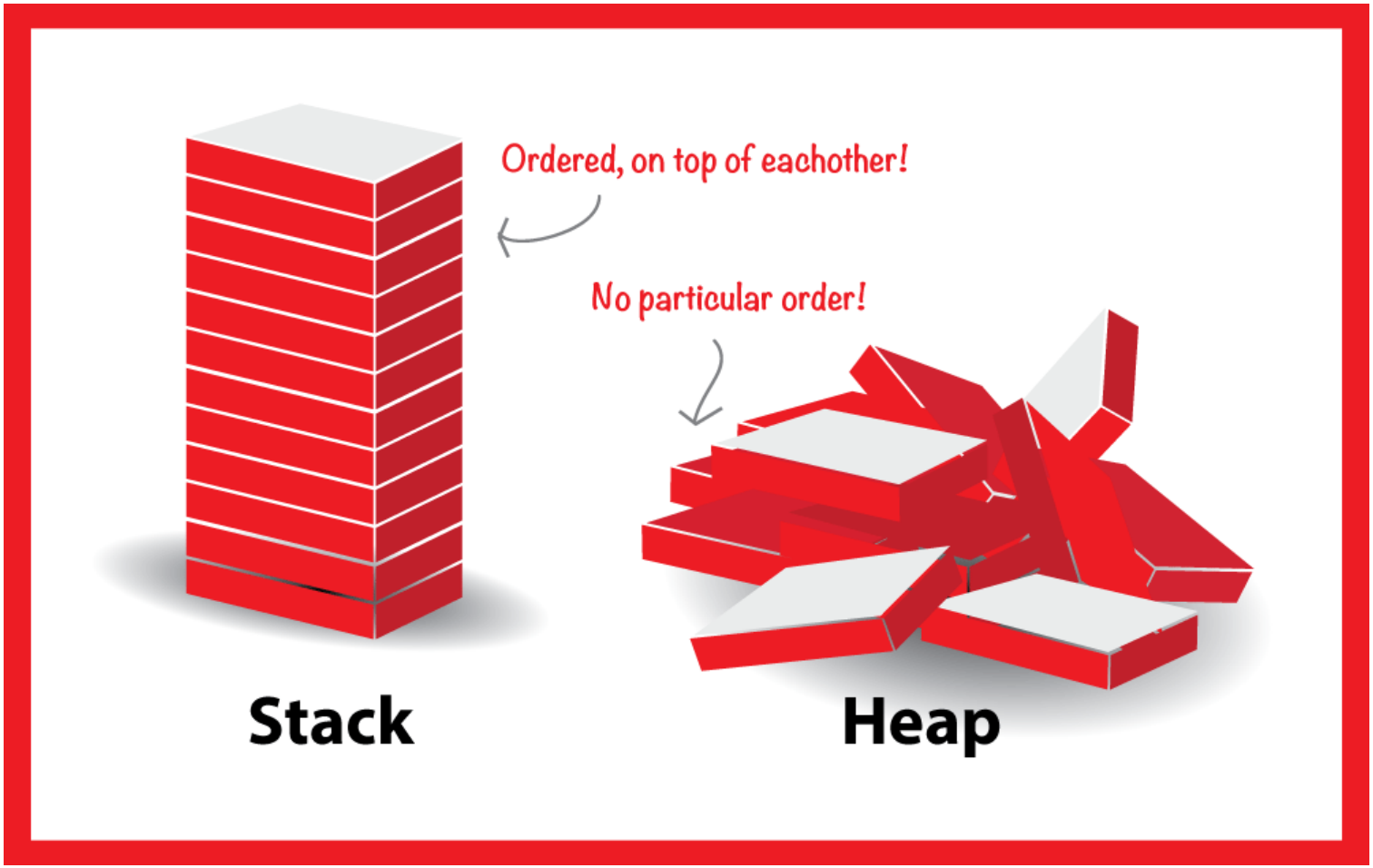
Less code ➡ Faster code

Computer Memory

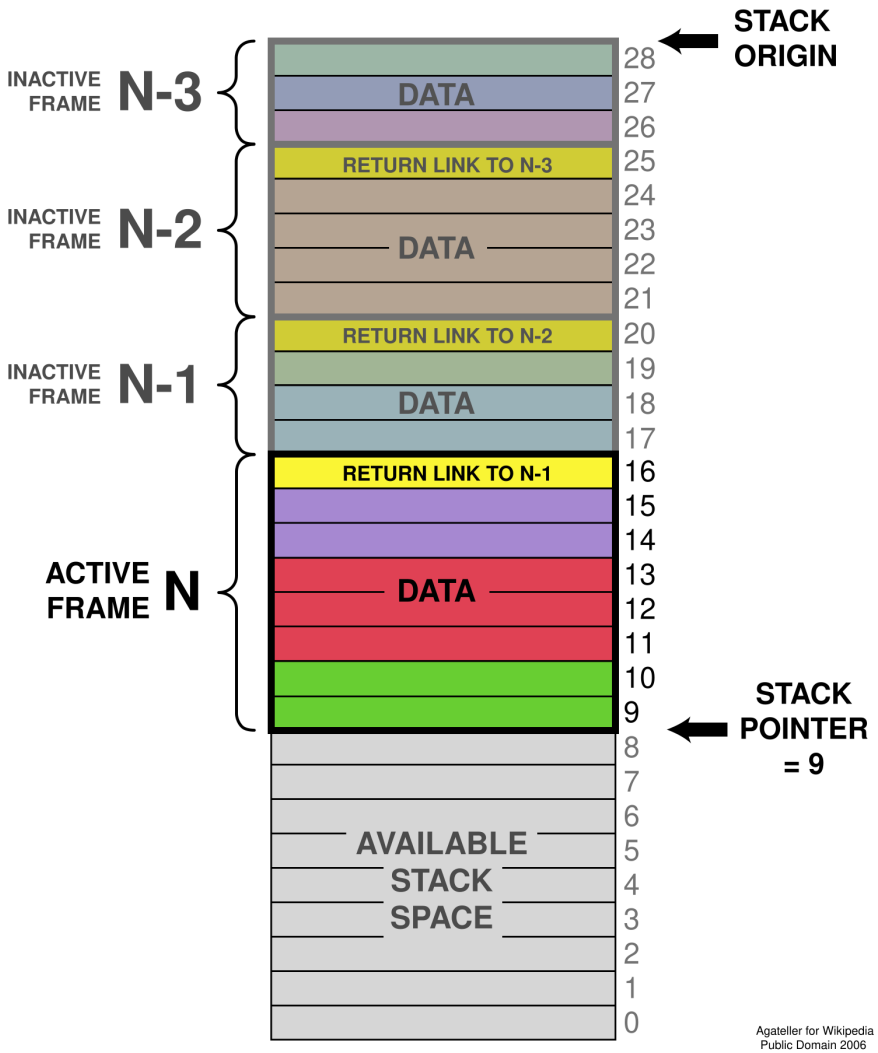


The Cost of Instructions

`double + double`	1 cycle
`double - double`	1 cycle
`double * double`	2 cycle
`double / double`	36 cycle
`size_t / size_t`	59 cycle
`size_t % size_t`	60 cycle
Read Register [~1kB]	0 cycle
Read L1 cache [192kB]	5 cycles
Read L2 cache [5MB]	10 cycles
Read L3 cache [12MB]	50 cycles
Read RAM [32GB]	200 cycles


Ordered, on top of eachother!
No particular order!
Stack
Heap

Call Stack



Agateller for Wikipedia
Public Domain 2006

Stack and Heap Variables

Variables on stack are basically for free

✔ Size known at compile-time

✔ On register or L1 cache

```
double x = 4;
std::array<double, 1024> a;
MyClass mc; // assuming no heap-allocated members
            // nor virtual functions
```

Variables on heap is expensive

x Allocation and deallocation costs

x Could be on RAM (cache-miss)

```
std::vector<double> v{1., 2.};
v.push_back(3.);
auto unique = std::make_unique<MyClass>();
auto shared = std::make_shared<MyClass>();
```

Indirection

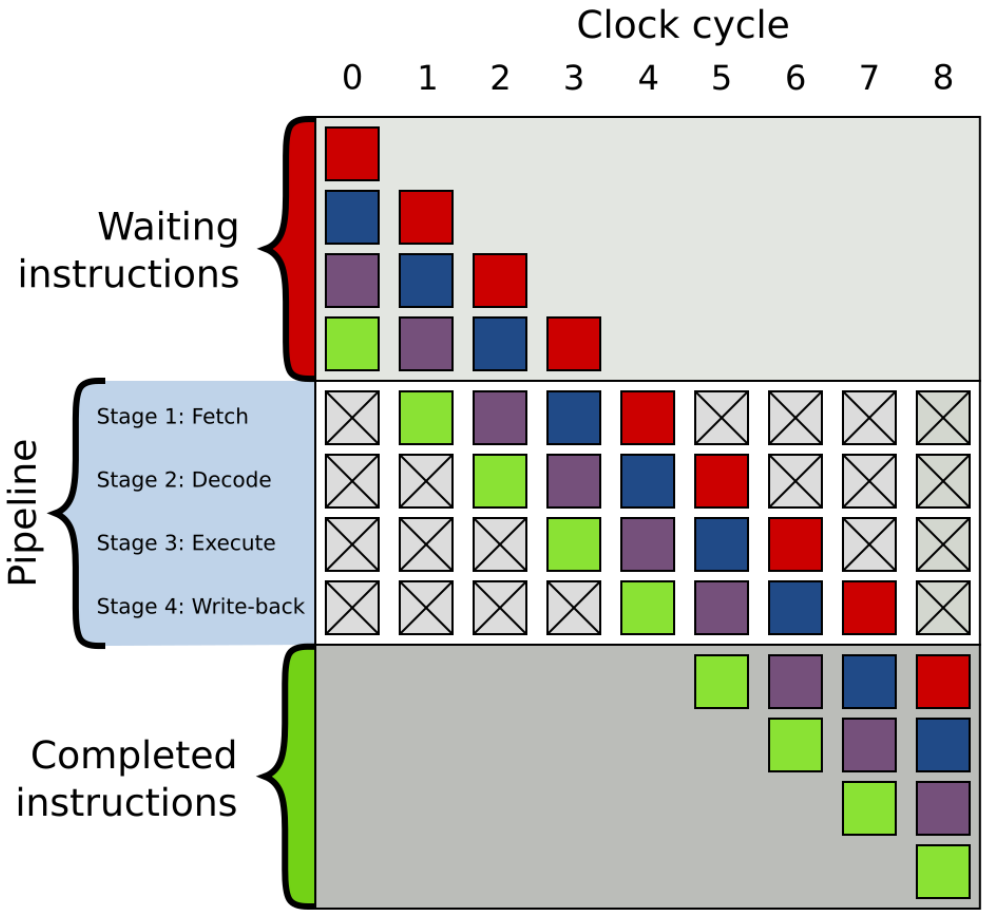
x Do not do this:

```
std::vector<std::vector<double>> a2d; // 2D Matrix
a2d.resize(10);
for(auto& a: a2d) a.resize(10);
a[3][4] = 5.;
```

✔ Do this:

```
size_t index2d(size_t i, size_t j, size_t N) {
    return i*N + j;
}
std::vector<double> a2d; // 2D Matrix
a2d.resize(10*10);
a[index2d(3, 4, 10)] = 5.;
```

Processor Pipeline



Branching

- (At least) 4 branches

```
for (auto other = Base::min; other < Base::max; ++other) {
    if (other == base.base)
        baseLikelihoods[other] = eps.complement();
    else
        baseLikelihoods[other] = (1./3) * eps;
}
```

Branching

✗ (At least) 4 branches

```
for (auto other = Base::min; other < Base::max; ++other) {
    if (other == base.base)
        baseLikelihoods[other] = eps.complement();
    else
        baseLikelihoods[other] = (1./3) * eps;
}
```

✔ No branching

```
baseLikelihoods.fill((1./3) * eps);
baseLikelihoods[base.base] = eps.complement();
```

Performance Considerations

✔ Measure!

x Do not micro-optimize

✔ Less code ➔ Faster code

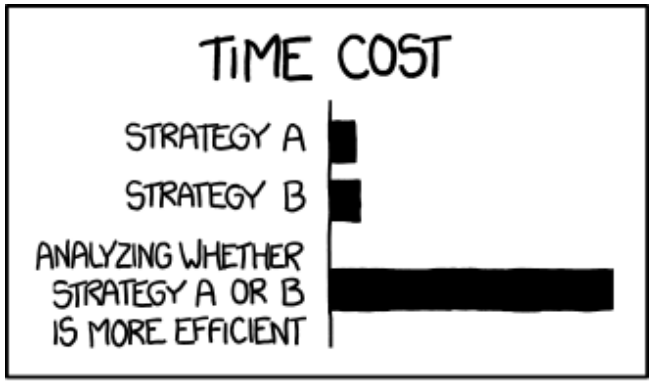
✔ Avoid heap-allocations

✔ Avoid indirection

✔ Avoid branching

Live C++ Example

`poly.cpp`


TIME COST
STRATEGY A
STRATEGY B
ANALYZING WHETHER STRATEGY A OR B IS MORE EFFICIENT
THE REASON I AM SO INEFFICIENT

Programming in C++

Why

- ✔ Old language, new features
- ✔ High performance code
- ✔ Reusable code

How

- ✔ Safety through modern features
- ✔ Be aware of performance costs
- ✔ Use available tools and resources