C/C++11 Memory model

What is it, and Why?

Who am I?

Mikael Rosbacke

Self employed Software consultant. Working in own company Akaza AB in cooperation with Berotec. Been doing electronics design earlier.

Main languages, C, C++, Python.

Main focus on Open source, IoT, Robotics, Linux, Embedded and (modern) microcontrollers.

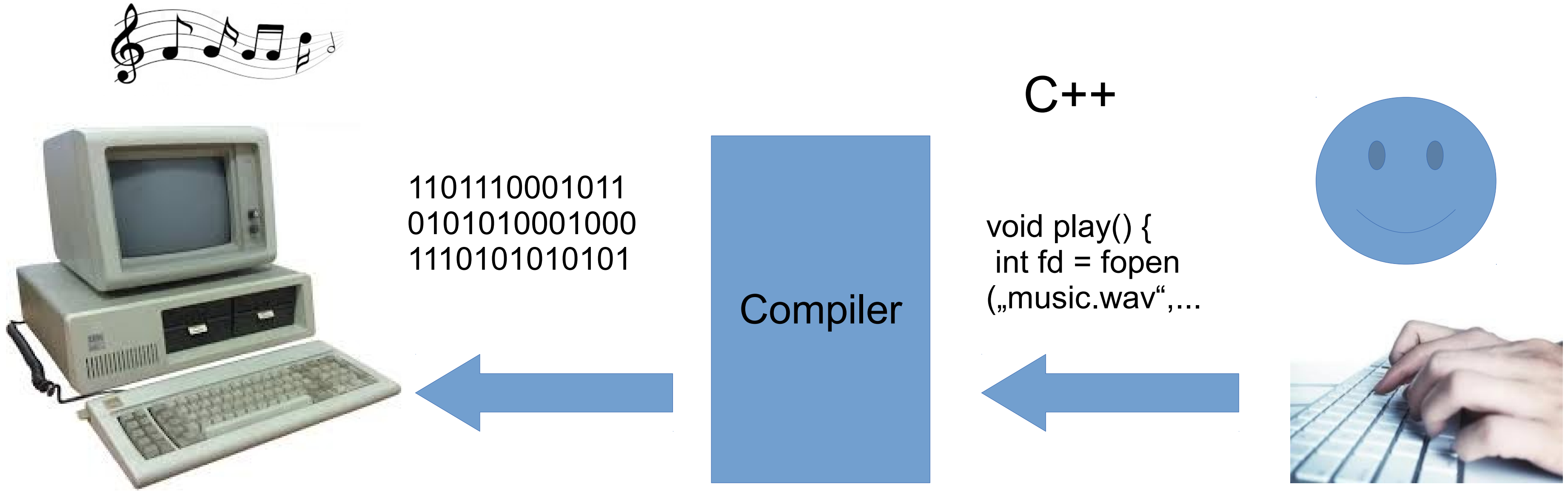
Started out at KTH. (M Sc EE, and some years doing doctoral studies in robotics and computer vision)

Current side projects:

- Power control: Allow third party users control your smart power outlets using a web browser.
- Serial_net: Talk TCP/IP over a shared serial bus (e.g. RS485).

http://akaza.se
http://ourcleanfuture.akaza.se
http://berotec.se
https://www.linkedin.com/in/mikael-rosbacke-13b8265

What is C++?

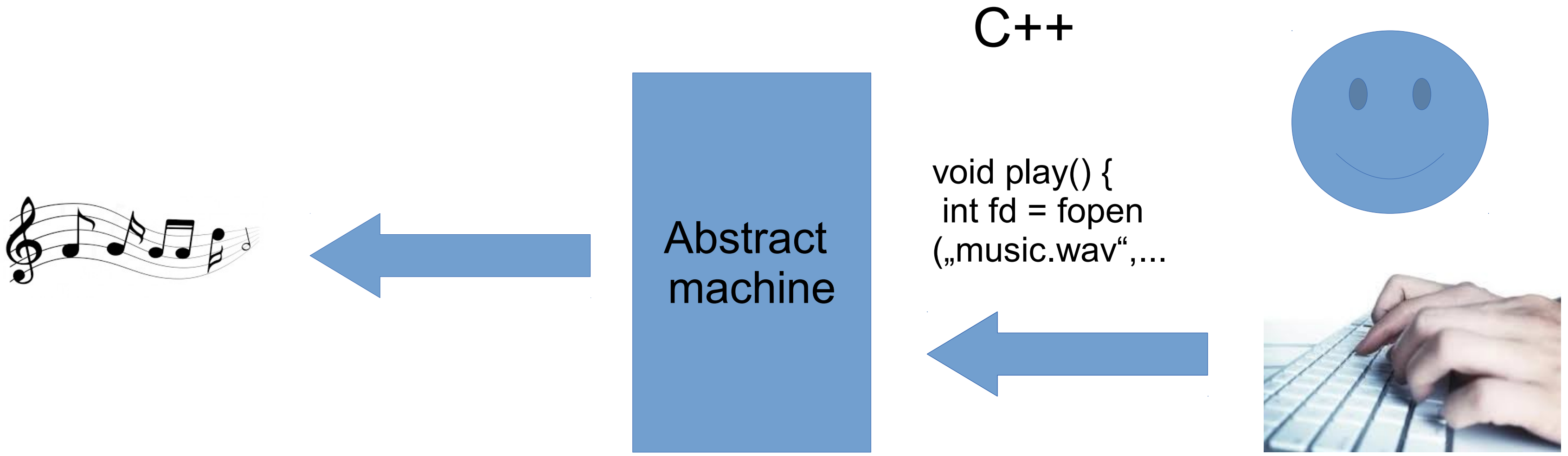


```
1101110001011
0101010001000
1110101010101
```

```
void play() {
 int fd = fopen
(„music.wav“,...
```

C++ is an <u>interface</u> between the programmer and the compiler writer. If you follow the rules for C++, the compiler writer promise to make the computer do your bidding. A <u>contract</u> is also a useful analogy.

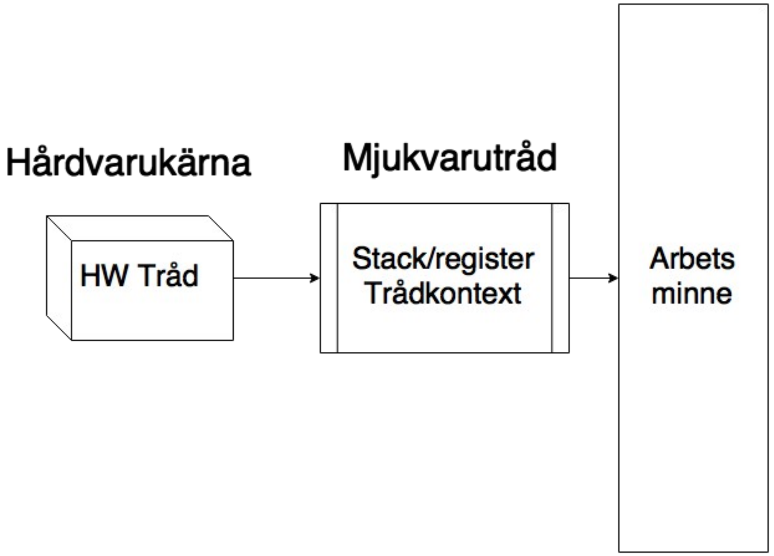
The abstract machine



- C++ is defined in terms of an abstract machine. You program toward this abstract machine. Most reasoning about the underlying hardware is outside of C++.
- The abstract machine is either implemented as hosted (full implementation, think full blown OS) or, freestanding (limited, think microcontrollers).
- C++ has inherited the abstract machine from C.

The early days.

C was developed in the 70ties. It was optimized for those machines.



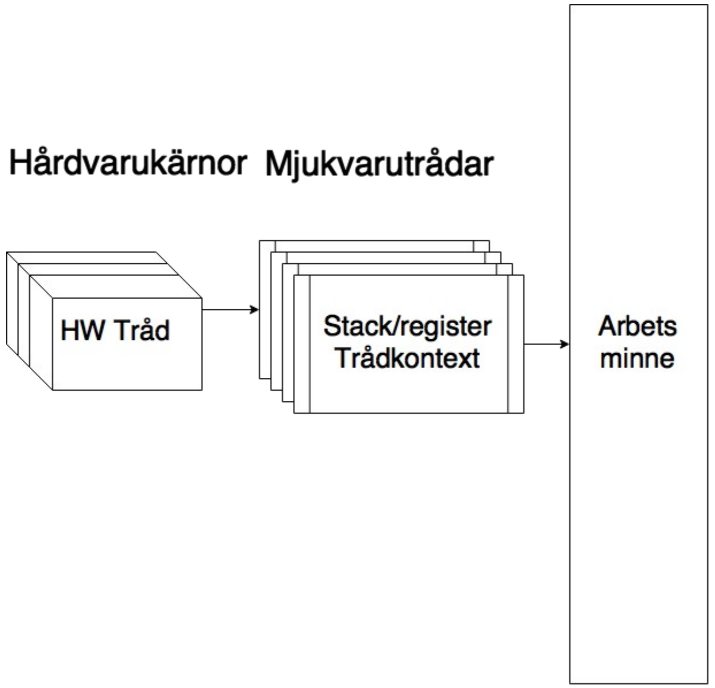
Processors where slow. All memory access took about the same time. (freq < 1MHz)

C was mainly developed for Unix. It is process heavy with a single thread for each process.

Threads (POSIX pthread) came way later in 1995.

The later days (2000)

C/C++ adopted MT via external libs. Same memory model as before. (pthreads, windows threads)



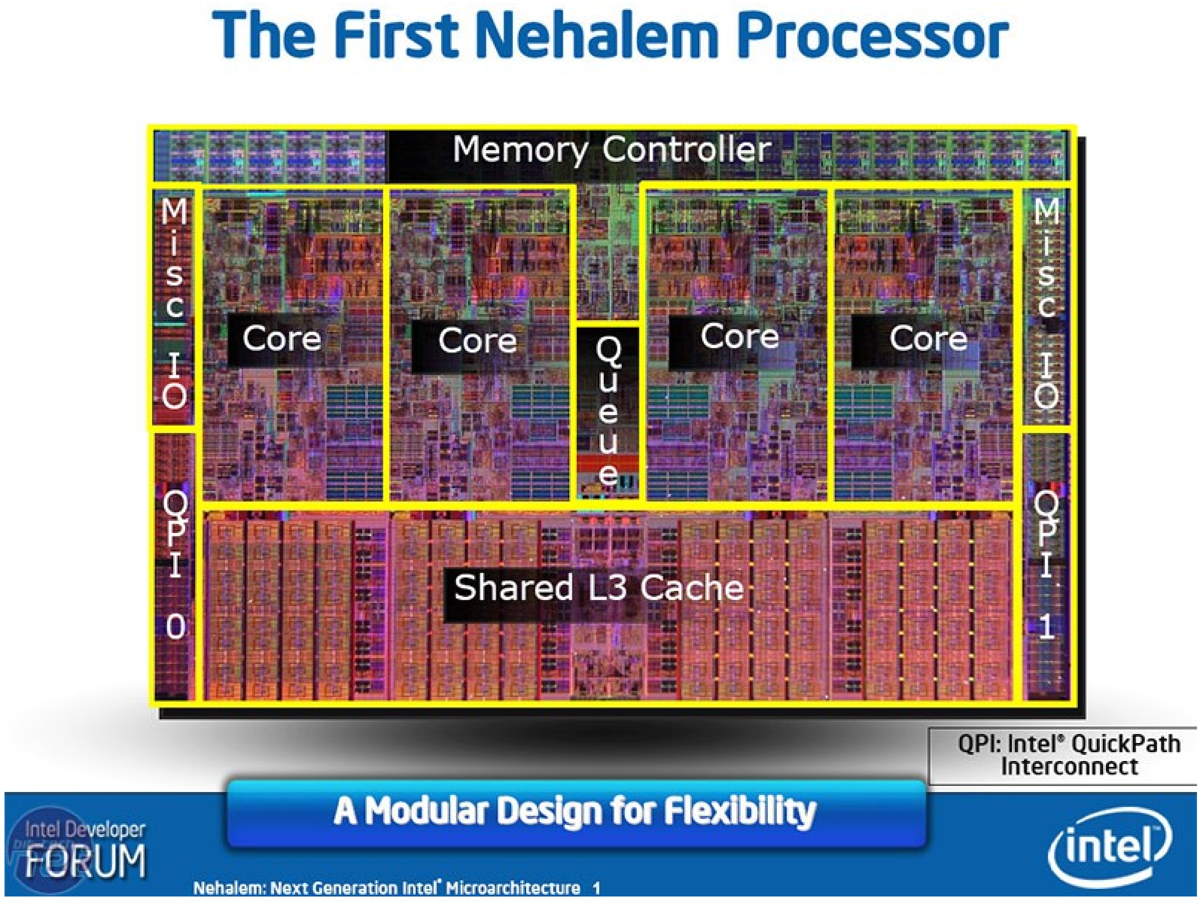
Hardware started supporting multiple execution cores/hyper threading.

CPU frequency skyrocketed.
Massive increase i amount of memory.
Memory latency got expensive (vs clock speed).
Cache hierarchies.

Still same abstract machine with same memory model. Bad fit for modern hardware. (C99 / C++98)

Modern hardware

Around 2009



The First Nehalem Processor
Memory Controller
Misc IO
Core
Core
Queue
Core
Core
Misc IO
QPI 0
Shared L3 Cache
QPI 1
QPI: Intel® QuickPath Interconnect
A Modular Design for Flexibility
Intel Developer FORUM
Nehalem: Next Generation Intel® Microarchitecture 1
intel

Multiple cores on die. We really want threads to use these.

Common L3 cache and memory controller on die. Individual L1, L2 caches for each core.

Access outside L1 via cachelines (e.g. 64 bytes), Need to sync content of caches.
Access to main memory >200 cycles.

Even the compiler has very limited control about timing of the memory system.
Memory system best described as packet based.

C++98 „as if“ rule

From cppreference:
Allows (the compiler to do) any and all code transformations that do not change the observable behavior of the program.

Only library calls and reading/writing volatile memory are observable in a ST program. Access to ordinary memory is not.
The compiler counts on this. It will happily remove read/writes, Insert extra writes and move accesses around.
Link time optimization will increase this.

Modern hardware assume extra instructions are given when memory is shared between cores. We need cooperation from the compiler.

Making sure all writes is immediately visible to all threads would be excruciatingly slow.

C11/C++11 memory model

Acc. to Wikipedia: A memory model describes the interactions of threads through memory and their shared use of the data.

Key here is interaction of threads. No memory model was needed with ST programs.

The memory model sets the rules that different threads needs to follow when <u>sharing</u> data in memory. Unshared data act as before.

Sidenote: Even with the new rules, You want your threads to be ‚shy‘. Do not share unneccessary data. Do not let data bounce back and forth between execution units. Think about the cache hierarchy.

C11/C++11 memory object

The standard defines an ‚object‘, here called memory object to avoid confusion with normal C++ objects.

A memory object is a region of memory with a specific type. (think memory of int, double, char etc)
Elements of an aggregate (arrays, structs etc) are separate memory objects.
(adjacent bitfields can share memory objects)

The standard gurantee that read / writes to different memory objects by different threads are independent. (The compiler must make that true).

Two different threads can read/write e.g. a[0] and a[1] without any issues.
MyStruct has 5 memory objects in it.

```
struct MyStruct
{
  char a[4];
  int b;
};
```

Sharing of data among threads

Sharing memory objects among threads (where one is a write) requires that:

- All accesses are atomic or,
- If any access is non-atomic, a ‚happens before‘ relationship must exist between the accesses.

If the above does not hold, we have undefined behavior.

So, what is ‚happens before‘?

Happens before

Short (simplified) version is that ‚happens before‘ relation is present when the compiler can prove that one operation is before another via a ‚synchronization event‘.
This is done using e.g. atomics with memory_order release (for writes) and acquire (for reads.) or mutexes.

Slightly longer answer: The notion of sequence points have been removed and is replaced by a family of operators called ‚happens before‘, ‚is_sequenced_before‘, ‚inter-thread happens before‘ and ‚synchronizes with‘. These are used to determine in what order observable behavior happens.

Release / Acquire

Two operations which pair up to form a synchronization.
(Allow ‚happens-before‘ between threads.)

Release : Issued together with a write operation.

Useful (but incorrect) mental model: A release instruct the compiler to ‚write out the data‘ to the main memory.

Acquire : Issued together with a read operation.

Useful (but incorrect) mental model: An acquire instruct the compiler to ‚gather the data‘ from main memory.

A write (release) in thread A matching up read (acquire) in thread B on a memory object, will introduce a sychronization.

→ Write in thread A ‚happens before‘ the read in thread B.

Atomics

New variable type. Two functions:

- Make an access all or nothing.
- Participates in synchronization.

```
#include <atomic>
using namespace std;

atomic<int> a_i;

a_i.store(1, memory_order_release);
int x = a_i.load(memory_order_acquire);
```

You can have atomics of e.g. classes.

There is no guarantee that an atomic is ‚lock_free‘. The compiler can issue hidden locks.

Memory order

Annotation to atomic operations.

seq_cst : Most restrictive, easiest to use, incurs most latency.

relaxed : no ordering, minimum latency.

A mutex ‚lock‘ does an ‚acquire‘.

A mutex ‚unlock‘ does a ‚release‘.

```
#include <atomic>
using namespace std;

atomic<int> a_i;

a_i.store(1, memory_order_release);
int x = a_i.load(memory_order_acquire);
```

Operation	load	store
sequential consistency	seq_cst	seq_cst
release/acquire	acquire	release
release	relaxed	release
acquire	acquire	relaxed
consume	consume	relaxed
relaxed	relaxed	relaxed

Minimum guarantee on how memory objects change

Assume you have a number of shared atomic memory objects. You only use ‚relaxed' to change these. What can we count on?

During the program run, each memory object has a number of values they will assume from initial value at start to the final value at the end. This order for a memory object is the same for all threads.

Each thread will never know which value it will read. All it knows is that it could be the same as the last, or an arbitrarily later value. It will never see an earlier value.

Different threads have different ‚pointers' into this order. One thread can read a late value while another is at the beginning. Also the order differs for different memory objects.

Summary

- The memory model mostly codifies earlier best practice w.r.t. MT. (threads, mutexes etc).
- The programmer can reason about memory objects as firm units when analyzing code.
- Hardware manufacturers uses this model when designing CPU ISA:s (e.g. ARMv8)
- If you have C++98 code using threads, consider upgrading to get the predictability of a known model.

In my view, the memory model strikes a good balance between backward compatibility to old code, while still taking modern hardware into account.

More reading

Herb Sutter material:

- atomic<> Weapons talk
  Search youtube

- The free lunch is over
  http://www.gotw.ca/publications/concurrency-ddj.htm

Anthony Williams: C++ Concurrency in Action

Your favourite bookstore. ISBN-10: 1933988770
The book that really drills down into the new model.

Standards:

C11 : ISO/IEC 9899:2011 Information technology - Programming Language C
(Public draft available from working group, WG14 N1570)
C++14 : ISO/IEC 14882:2014(E) – Programming Language C++
(Public draft available as free pdf)