SQUAREHEAD TECHNOLOGY

THE TWO MEMORY MODELS

ANDERS SCHAU KNATTEN

Squarehead / CppQuiz.org

MEETING C++ 2025

SQUAREHEAD
TECHNOLOGY

SQUAREHEAD TECHNOLOGY

CPPQUIZ.ORG

C++ Quiz

You've answered 0 of 178 questions correctly. (Clear)

Question #197 Difficulty: ●○○

According to the C++23 standard, what is the output of this program?

```
#include <iostream>

int j = 1;

int main() {
  int& i = j, j;
  j = 2;
  std::cout << i << j;
}
```

Answer: The program is guaranteed to output: Answer

Problems? View a hint or try another question.

I give up, show me the answer (make 3 more attempts first).

Mode : Training

You are currently in training mode, answering random questions. Why not Start a new quiz? Then you can boast about your score, and invite your friends.

Contribute

Create your own!

Android app

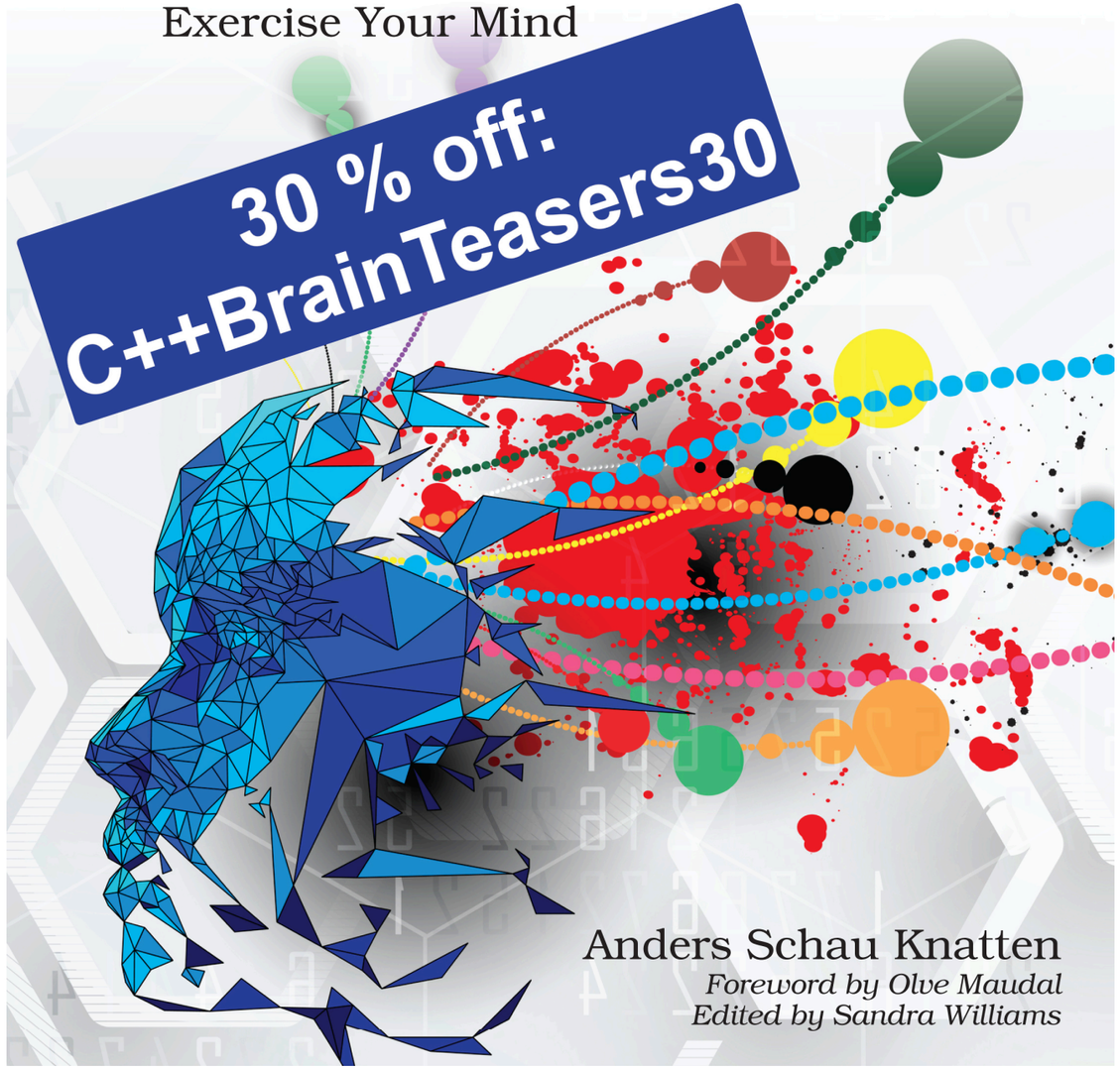
Get Sergey Vasilchenko's CppQuiz Android app.

SQUAREHEAD TECHNOLOGY

The Pragmatic Programmers

C++ Brain Teasers

Exercise Your Mind



SQUAREHEAD
TECHNOLOGY

THE TWO MEMORY MODELS

THE TWO MEMORY MODELS

- Why?

SQUAREHEAD TECHNOLOGY

THE TWO MEMORY MODELS

- Why?
- Theoretical background

SQUAREHEAD TECHNOLOGY

THE TWO MEMORY MODELS

- Why?
- Theoretical background
- Intuition, framework for reasoning

SQUAREHEAD TECHNOLOGY

THE TWO MEMORY MODELS

- Why?
- Theoretical background
- Intuition, framework for reasoning
- How stuff works!

SQUAREHEAD TECHNOLOGY

THE TWO MEMORY MODELS

THE TWO MEMORY MODELS

- High Level language (C++, C, Rust)

THE TWO MEMORY MODELS

- High Level language (C++, C, Rust)
- CPU / Architecture

SQUAREHEAD
TECHNOLOGY

C++ MEMORY MODEL

SQUAREHEAD TECHNOLOGY

C++ MEMORY MODEL

6.7.1 Memory model [intro.memory]

1 The fundamental storage unit in the C++ memory model is the byte. A byte is at least large enough to contain any member of the execution character set ([lex.charset]) and the eight-bit code units of the Unicode[89] UTF-8 encoding form and is composed of a contiguous sequence of bits,[33] the number of which is implementation-defined. The least significant bit is called the *low-order bit*; the most significant bit is called the *high-order bit*. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address.

2 A *memory location* is either an object of scalar type that is not a bit-field or a maximal sequence of adjacent bit-fields all having nonzero width. Two or more threads of execution can access separate memory locations without interfering with each other.

SQUAREHEAD TECHNOLOGY

C++ MEMORY MODEL

6.7.1 **Memory model** [intro.memory]

1 The fundamental storage unit in the C++ memory model is the byte. A byte is at least large enough to contain any member of the execution character set ([lex.charset]) and the eight-bit code units of the Unicode[90] UTF-8 encoding form and is composed of a contiguous sequence of bits,[91] the number of which is implementation-defined. The least significant bit is called the *low-order bit*; the most significant bit is called the *high-order bit*. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address.

2 A *memory location* is either an object of scalar type that is not a bit-field or a maximal sequence of adjacent bit-fields all having nonzero width. Two or more threads of execution can access separate memory locations without interfering with each other.

- What's meant by "a memory location"

SQUAREHEAD TECHNOLOGY

C++ MEMORY MODEL

6.7.1 **Memory model** [intro.memory]

1 The fundamental storage unit in the C++ memory model is the byte. A byte is at least large enough to contain any member of the execution character set ([lex.charset]) and the eight-bit code units of the Unicode[89] UTF-8 encoding form and is composed of a contiguous sequence of bits,[53] the number of which is implementation-defined. The least significant bit is called the *low-order bit*; the most significant bit is called the *high-order bit*. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address.

2 A *memory location* is either an object of scalar type that is not a bit-field or a maximal sequence of adjacent bit-fields all having nonzero width. Two or more threads of execution can access separate memory locations without interfering with each other.

- What's meant by "a memory location"
- Two or more threads can access separate memory locations without interfering with each other.

C++ MEMORY MODEL

- But other rules too:

C++ MEMORY MODEL

- But other rules too:
  - Sequential execution [intro.execution]

C++ MEMORY MODEL

- But other rules too:
  - Sequential execution [intro.execution]
  - Multi-threaded executions and data races [intro.multithread]

C++ MEMORY MODEL

- But other rules too:
  - Sequential execution [intro.execution]
  - Multi-threaded executions and data races [intro.multithread]
  - Atomic operations library [atomics]

C++ MEMORY MODEL

- But other rules too:
  - Sequential execution [intro.execution]
  - Multi-threaded executions and data races [intro.multithread]
  - Atomic operations library [atomics]
  - Thread support library [thread]

SQUAREHEAD
TECHNOLOGY

CPU MEMORY MODEL

CPU MEMORY MODEL

- X86, ARM, RISC-V, ...

CPU MEMORY MODEL

- X86, ARM, RISC-V, ...
- Described in the architecture manual

CPU MEMORY MODEL

- X86, ARM, RISC-V, ...
- Described in the architecture manual
- "A memory consistency model is a set of rules specifying the values that can be returned by loads of memory." - RISC-V ISA manual

SQUAREHEAD
TECHNOLOGY

CPU MEMORY MODEL

CPU MEMORY MODEL

- Simple if you have one thread

CPU MEMORY MODEL

- Simple if you have one thread
  - A load from an address can't observe a later store to the same

CPU MEMORY MODEL

- Simple if you have one thread
  - A load from an address can't observe a later store to the same
  - Program + input state = output state

CPU MEMORY MODEL

- Simple if you have one thread
  - A load from an address can't observe a later store to the same
  - Program + input state = output state
- Complicated if you have many threads

CPU MEMORY MODEL

- Simple if you have one thread
  - A load from an address can't observe a later store to the same
  - Program + input state = output state
- Complicated if you have many threads
  - Any number of valid interleavings of threads

CPU MEMORY MODEL

- Simple if you have one thread
  - A load from an address can't observe a later store to the same
  - Program + input state = output state
- Complicated if you have many threads
  - Any number of valid interleavings of threads
  - Program + input state = many possible output states

SQUAREHEAD TECHNOLOGY

WHAT HAS THE MEMORY MODEL EVER DONE FOR ME?

SQUAREHEAD TECHNOLOGY

WHAT HAS THE MEMORY MODEL EVER DONE FOR ME?

- C++ memory model
  - Single threaded examples
  - Multi threaded example
  - Memory ordering

WHAT HAS THE MEMORY MODEL EVER DONE FOR ME?

- C++ memory model
  - Single threaded examples
  - Multi threaded example
  - Memory ordering
- CPU memory model:
  - Several variations, including x86 and RISC-V

SINGLE THREADED EXAMPLE

SINGLE THREADED EXAMPLE

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;
4     result += 1;
5     result += *b;
6
7     return result;
8 }
```

SINGLE THREADED EXAMPLE

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;
4     result += 1;
5     result += *b;
6
7     return result;
8 }
```

SINGLE THREADED EXAMPLE

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;
4     result += 1;
5     result += *b;
6
7     return result;
8 }
```

SINGLE THREADED EXAMPLE

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;
4     result += 1;
5     result += *b;
6
7     return result;
8 }
```

SINGLE THREADED EXAMPLE

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;
4     result += 1;
5     result += *b;
6
7     return result;
8 }
```

SINGLE THREADED EXAMPLE

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;     // load reg_0, *a
4     result += 1;         // add  reg_0, reg_0, 1
5     result += *b;        // load reg_1, *b
6                          // add  reg_0, reg_0, reg_1
7     return result;       // ret  reg_0
8 }
```

SINGLE THREADED EXAMPLE

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;     // load reg_0, *a
4     result += 1;         // add  reg_0, reg_0, 1
5     result += *b;        // load reg_1, *b
6                          // add  reg_0, reg_0, reg_1
7     return result;       // ret  reg_0
8 }
```

SINGLE THREADED EXAMPLE

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;     // load reg_0, *a
4     result += 1;         // add  reg_0, reg_0, 1
5     result += *b;        // load reg_1, *b
6                          // add  reg_0, reg_0, reg_1
7     return result;       // ret  reg_0
8 }
```

SINGLE THREADED EXAMPLE

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;     // load reg_0, *a
4     result += 1;         // add  reg_0, reg_0, 1
5     result += *b;        // load reg_1, *b
6                          // add  reg_0, reg_0, reg_1
7     return result;       // ret  reg_0
8 }
```

SINGLE THREADED EXAMPLE

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;     // load reg_0, *a
4     result += 1;         // add  reg_0, reg_0, 1
5     result += *b;        // load reg_1, *b
6                          // add  reg_0, reg_0, reg_1
7     return result;       // ret  reg_0
8 }
```

SINGLE THREADED EXAMPLE

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;     // load reg_0, *a
4     result += 1;         // add  reg_0, reg_0, 1
5     result += *b;        // load reg_1, *b
6                          // add  reg_0, reg_0, reg_1
7     return result;       // ret  reg_0
8 }
```

SQUAREHEAD TECHNOLOGY

C++: "SEQUENCED BEFORE"

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;     // load reg_0, *a
4     result += 1;         // add  reg_0, reg_0, 1
5     result += *b;        // load reg_1, *b
6                          // add  reg_0, reg_0, reg_1
7     return result;       // ret  reg_0
8 }
```

SQUAREHEAD TECHNOLOGY

C++: "SEQUENCED BEFORE"

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;     // load reg_0, *a
4     result += 1;         // add  reg_0, reg_0, 1
5     result += *b;        // load reg_1, *b
6                          // add  reg_0, reg_0, reg_1
7     return result;       // ret  reg_0
8 }
```

- Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.

SQUAREHEAD TECHNOLOGY

C++: "SEQUENCED BEFORE"

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;     // load reg_0, *a
4     result += 1;         // add  reg_0, reg_0, 1
5     result += *b;        // load reg_1, *b
6                          // add  reg_0, reg_0, reg_1
7     return result;       // ret  reg_0
8 }
```

- Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.
- Standard does not dictate how to *implement* C++!

SQUAREHEAD TECHNOLOGY

C++: "SEQUENCED BEFORE"

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;     // load reg_0, *a
4     result += 1;         // add  reg_0, reg_0, 1
5     result += *b;        // load reg_1, *b
6                          // add  reg_0, reg_0, reg_1
7     return result;       // ret  reg_0
8 }
```

- Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.
- Standard does not dictate how to *implement* C++!
- Abstract machine

SQUAREHEAD TECHNOLOGY

C++: "SEQUENCED BEFORE"

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;     // load reg_0, *a
4     result += 1;         // add  reg_0, reg_0, 1
5     result += *b;        // load reg_1, *b
6                          // add  reg_0, reg_0, reg_1
7     return result;       // ret  reg_0
8 }
```

- Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.
- Standard does not dictate how to *implement* C++!
- Abstract machine
- Only need to emulate observable behaviour

SQUAREHEAD TECHNOLOGY

C++: "SEQUENCED BEFORE"

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;     // load reg_0, *a
4     result += 1;         // add  reg_0, reg_0, 1
5     result += *b;        // load reg_1, *b
6                          // add  reg_0, reg_0, reg_1
7     return result;       // ret  reg_0
8 }
```

- Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.
- Standard does not dictate how to *implement* C++!
- Abstract machine
- Only need to emulate observable behaviour
- Observable: I/O, files, volatile, plus more implementation defined

SQUAREHEAD TECHNOLOGY

C++: "SEQUENCED BEFORE"

```
1 int compute(const int* a, const int* b)
2 {
3     int result = *a;     // load reg_0, *a
4     result += 1;         // add  reg_0, reg_0, 1
5     result += *b;        // load reg_1, *b
6                          // add  reg_0, reg_0, reg_1
7     return result;       // ret  reg_0
8 }
```

- Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.
- Standard does not dictate how to *implement* C++!
- Abstract machine
- Only need to emulate observable behaviour
- Observable: I/O, files, volatile, plus more implementation defined
- Can reorder

SINGLE THREADED EXAMPLE

```
1      // original                    // reordered
2      load reg_0, *a                 load reg_0, *a
3      add  reg_0, reg_0, 1           load reg_1, *b
4      load reg_1, *b                 add  reg_0, reg_0, 1
5      add  reg_0, reg_0, reg_1       add  reg_0, reg_0, reg_1
6      ret  reg_0                     ret  reg_0
```

SINGLE THREADED EXAMPLE

```
1      // original                    // reordered
2      load reg_0, *a                 load reg_0, *a
3      add  reg_0, reg_0, 1           load reg_1, *b
4      load reg_1, *b                 add  reg_0, reg_0, 1
5      add  reg_0, reg_0, reg_1       add  reg_0, reg_0, reg_1
6      ret  reg_0                     ret  reg_0
```

SINGLE THREADED EXAMPLE

```
1      // original                    // reordered
2      load reg_0, *a                 load reg_0, *a
3      add  reg_0, reg_0, 1           load reg_1, *b
4      load reg_1, *b                 add  reg_0, reg_0, 1
5      add  reg_0, reg_0, reg_1       add  reg_0, reg_0, reg_1
6      ret  reg_0                     ret  reg_0
```

- Or CPU reorders it on the fly

SINGLE THREADED EXAMPLE

```
1      // original                    // reordered
2      load reg_0, *a                 load reg_0, *a
3      add  reg_0, reg_0, 1           load reg_1, *b
4      load reg_1, *b                 add  reg_0, reg_0, 1
5      add  reg_0, reg_0, reg_1       add  reg_0, reg_0, reg_1
6      ret  reg_0                     ret  reg_0
```

- Or CPU reorders it on the fly
- "All" modern CPUs are out-of-order

SINGLE THREADED EXAMPLE

```
1     // original                    // reordered
2     load reg_0, *a                 load reg_0, *a
3     add  reg_0, reg_0, 1           load reg_1, *b
4     load reg_1, *b                 add  reg_0, reg_0, 1
5     add  reg_0, reg_0, reg_1       add  reg_0, reg_0, reg_1
6     ret  reg_0                     ret  reg_0
```

- Or CPU reorders it on the fly
- "All" modern CPUs are out-of-order
- Can we swap the order of the loads too?

SINGLE THREADED EXAMPLE

```
1      // original                    // reordered
2      load reg_0, *a                 load reg_0, *a
3      add  reg_0, reg_0, 1           load reg_1, *b
4      load reg_1, *b                 add  reg_0, reg_0, 1
5      add  reg_0, reg_0, reg_1       add  reg_0, reg_0, reg_1
6      ret  reg_0                     ret  reg_0
```

- Or CPU reorders it on the fly
- "All" modern CPUs are out-of-order
- Can we swap the order of the loads too?
- Yes, can't affect the observable behaviour

SINGLE THREADED EXAMPLE (X86)

GCC 15.1 –03

```
int compute(const int* a, const int* b)
{
    int result = *a;
    result += 1;
    result += *b;
    return result;
}
```

```
1 compute(int const*, int const*):
2 mov     edx, DWORD PTR [rdi]
3 mov     eax, DWORD PTR [rsi]
4 lea     eax, [rdx+1+rax]
5 ret
```

SINGLE THREADED EXAMPLE (X86)

GCC 15.1 –03

```
int compute(const int* a, const int* b)
{
    int result = *a;
    result += 1;
    result += *b;
    return result;
}
```

```
1 compute(int const*, int const*):
2 mov     edx, DWORD PTR [rdi]
3 mov     eax, DWORD PTR [rsi]
4 lea     eax, [rdx+1+rax]
5 ret
```

SINGLE THREADED EXAMPLE (X86)

GCC 15.1 –03

```
int compute(const int* a, const int* b)
{
    int result = *a;
    result += 1;
    result += *b;
    return result;
}
```

```
1 compute(int const*, int const*):
2 mov     edx, DWORD PTR [rdi]
3 mov     eax, DWORD PTR [rsi]
4 lea     eax, [rdx+1+rax]
5 ret
```

SINGLE THREADED EXAMPLE 2

```
1 void compute(int* a, int* b)
2 {
3     int result = get_value();
4     result++;
5     *a = 2;
6     *b = result;
7 }
```

SINGLE THREADED EXAMPLE 2

```
1 void compute(int* a, int* b)
2 {
3     int result = get_value();
4     result++;
5     *a = 2;
6     *b = result;
7 }
```

SINGLE THREADED EXAMPLE 2

```
1 void compute(int* a, int* b)
2 {
3     int result = get_value();
4     result++;
5     *a = 2;
6     *b = result;
7 }
```

SINGLE THREADED EXAMPLE 2

```
1 void compute(int* a, int* b)
2 {
3     int result = get_value();
4     result++;
5     *a = 2;
6     *b = result;
7 }
```

SINGLE THREADED EXAMPLE 2

```
1 void compute(int* a, int* b)
2 {
3     int result = get_value();
4     result++;
5     *a = 2;
6     *b = result;
7 }
```

SINGLE THREADED EXAMPLE 2

```
1 void compute(int* a, int* b)
2 {
3     int result = get_value();    // move  reg_0, get_value()
4     result++;                    // add   reg_0, reg_0, 1
5     *a = 2;                      // store *a, 2
6     *b = result;                 // store *b, reg_0
7 }
```

SINGLE THREADED EXAMPLE 2

```
1 void compute(int* a, int* b)
2 {
3     int result = get_value();    // move  reg_0, get_value()
4     result++;                    // add   reg_0, reg_0, 1
5     *a = 2;                      // store *a, 2
6     *b = result;                 // store *b, reg_0
7 }
```

SINGLE THREADED EXAMPLE 2

```
1 void compute(int* a, int* b)
2 {
3     int result = get_value();    // move  reg_0, get_value()
4     result++;                    // add   reg_0, reg_0, 1
5     *a = 2;                      // store *a, 2
6     *b = result;                 // store *b, reg_0
7 }
```

- Can we start the store to a earlier?

SINGLE THREADED EXAMPLE 2

```
1     // original
2     move  reg_0, get_value()
3     add   reg_0, reg_0, 1
4     store *a, 2
5     store *b, reg_0
```

SINGLE THREADED EXAMPLE 2

```
1     // original
2     move  reg_0, get_value()
3     add   reg_0, reg_0, 1
4     store *a, 2
5     store *b, reg_0
```

SQUAREHEAD TECHNOLOGY

SINGLE THREADED EXAMPLE 2

```
1     // original
2     move  reg_0, get_value()
3     add   reg_0, reg_0, 1
4     store *a, 2
5     store *b, reg_0
```

```
1     // reordered
2     move  reg_0, get_value()
3     store *a, 2
4     add   reg_0, reg_0, 1
5     store *b, reg_0
```

SINGLE THREADED EXAMPLE 2

```
1     // original
2     move  reg_0, get_value()
3     add   reg_0, reg_0, 1
4     store *a, 2
5     store *b, reg_0
```

```
1     // reordered
2     move  reg_0, get_value()
3     store *a, 2
4     add   reg_0, reg_0, 1
5     store *b, reg_0
```

SINGLE THREADED EXAMPLE 2

```
1     // original
2     move  reg_0, get_value()
3     add   reg_0, reg_0, 1
4     store *a, 2
5     store *b, reg_0
```

```
1     // reordered
2     move  reg_0, get_value()
3     store *a, 2
4     add   reg_0, reg_0, 1
5     store *b, reg_0
```

SINGLE THREADED EXAMPLE 2

```
1     // original
2     move  reg_0, get_value()
3     add   reg_0, reg_0, 1
4     store *a, 2
5     store *b, reg_0
```

```
1     // reordered
2     move  reg_0, get_value()
3     store *a, 2
4     add   reg_0, reg_0, 1
5     store *b, reg_0
```

SINGLE THREADED EXAMPLE 2

```
1     // original
2     move  reg_0, get_value()
3     add   reg_0, reg_0, 1
4     store *a, 2
5     store *b, reg_0
```

```
1     // reordered
2     move  reg_0, get_value()
3     store *a, 2
4     add   reg_0, reg_0, 1
5     store *b, reg_0
```

Or CPU reorders it

SINGLE THREADED EXAMPLE 2

```
1     // original
2     move  reg_0, get_value()
3     add   reg_0, reg_0, 1
4     store *a, 2
5     store *b, reg_0
```

```
1     // reordered
2     move  reg_0, get_value()
3     store *a, 2
4     add   reg_0, reg_0, 1
5     store *b, reg_0
```

Or CPU reorders it

CPU could reorder the stores!

SINGLE THREADED EXAMPLE 2

```
move  reg_0, get_value()
store *a, 2
add   reg_0, reg_0, 1
store *b, reg_0
```

SINGLE THREADED EXAMPLE 2

```
move  reg_0, get_value()
store *a, 2
add   reg_0, reg_0, 1
store *b, reg_0
```

- C++ to CPU: Do not store a before calling get_value!

SINGLE THREADED EXAMPLE 2

```
move  reg_0, get_value()
store *a, 2
add   reg_0, reg_0, 1
store *b, reg_0
```

- C++ to CPU: Do not store a before calling get_value!
- C++ to CPU: Do not reorder stores!

SINGLE THREADED EXAMPLE 2

```
move  reg_0, get_value()
store *a, 2
add   reg_0, reg_0, 1
store *b, reg_0
```

- C++ to CPU: Do not store a before calling get_value!
- C++ to CPU: Do not reorder stores!
- How do we express this to the CPU?

SQUAREHEAD TECHNOLOGY

SINGLE THREADED EXAMPLE 2 (X86)

GCC 15.1 –03

```
void compute(int* a, int* b)
{
    int result = get_value();
    result++;
    *a = 2;
    *b = result;
}
```

```
 1 compute(int*, int*):
 2 push    rbp
 3 mov     rbp, rdi
 4 push    rbx
 5 mov     rbx, rsi
 6 sub     rsp, 8
 7 call    get_value()
 8 mov     DWORD PTR [rbp+0], 2
 9 add     eax, 1
10 mov     DWORD PTR [rbx], eax
11 add     rsp, 8
12 pop     rbx
13 pop     rbp
14 ret
```

SQUAREHEAD TECHNOLOGY

SINGLE THREADED EXAMPLE 2 (X86)

GCC 15.1 –03

```
void compute(int* a, int* b)
{
    int result = get_value();
    result++;
    *a = 2;
    *b = result;
}
```

```
 1 compute(int*, int*):
 2 push    rbp
 3 mov     rbp, rdi
 4 push    rbx
 5 mov     rbx, rsi
 6 sub     rsp, 8
 7 call    get_value()
 8 mov     DWORD PTR [rbp+0], 2
 9 add     eax, 1
10 mov     DWORD PTR [rbx], eax
11 add     rsp, 8
12 pop     rbx
13 pop     rbp
14 ret
```

SQUAREHEAD TECHNOLOGY

SINGLE THREADED EXAMPLE 2 (X86)

GCC 15.1 –03

```
void compute(int* a, int* b)
{
    int result = get_value();
    result++;
    *a = 2;
    *b = result;
}
```

```
 1 compute(int*, int*):
 2 push    rbp
 3 mov     rbp, rdi
 4 push    rbx
 5 mov     rbx, rsi
 6 sub     rsp, 8
 7 call    get_value()
 8 mov     DWORD PTR [rbp+0], 2
 9 add     eax, 1
10 mov     DWORD PTR [rbx], eax
11 add     rsp, 8
12 pop     rbx
13 pop     rbp
14 ret
```

- X86 never reorders stores

SQUAREHEAD TECHNOLOGY

SINGLE THREADED EXAMPLE 2 (X86)

GCC 15.1 –03

```
void compute(int* a, int* b)
{
    int result = get_value();
    result++;
    *a = 2;
    *b = result;
}
```

```
 1 compute(int*, int*):
 2 push    rbp
 3 mov     rbp, rdi
 4 push    rbx
 5 mov     rbx, rsi
 6 sub     rsp, 8
 7 call    get_value()
 8 mov     DWORD PTR [rbp+0], 2
 9 add     eax, 1
10 mov     DWORD PTR [rbx], eax
11 add     rsp, 8
12 pop     rbx
13 pop     rbp
14 ret
```

- X86 never reorders stores
- RISC-V **can** reorder stores

SQUAREHEAD TECHNOLOGY

SINGLE THREADED EXAMPLE 2 (X86)

GCC 15.1 –03

```
void compute(int* a, int* b)
{
    int result = get_value();
    result++;
    *a = 2;
    *b = result;
}
```

```
 1 compute(int*, int*):
 2 push    rbp
 3 mov     rbp, rdi
 4 push    rbx
 5 mov     rbx, rsi
 6 sub     rsp, 8
 7 call    get_value()
 8 mov     DWORD PTR [rbp+0], 2
 9 add     eax, 1
10 mov     DWORD PTR [rbx], eax
11 add     rsp, 8
12 pop     rbx
13 pop     rbp
14 ret
```

- X86 never reorders stores
- RISC-V **can** reorder stores *to different addresses*

SQUAREHEAD TECHNOLOGY

SINGLE THREADED EXAMPLE 2 (X86)

GCC 15.1 –03

```
void compute(int* a, int* b)
{
    int result = get_value();
    result++;
    *a = 2;
    *b = result;
}
```

```
 1 compute(int*, int*):
 2 push    rbp
 3 mov     rbp, rdi
 4 push    rbx
 5 mov     rbx, rsi
 6 sub     rsp, 8
 7 call    get_value()
 8 mov     DWORD PTR [rbp+0], 2
 9 add     eax, 1
10 mov     DWORD PTR [rbx], eax
11 add     rsp, 8
12 pop     rbx
13 pop     rbp
14 ret
```

- X86 never reorders stores
- RISC-V **can** reorder stores *to different addresses*
- Reorder includes effects from pipeline, caches, store buffers etc

SQUAREHEAD
TECHNOLOGY

MULTI THREADED EXAMPLE

SQUAREHEAD TECHNOLOGY

MULTI THREADED EXAMPLE

SQUAREHEAD TECHNOLOGY

MULTI THREADED EXAMPLE

```
1  Data data;
2  bool ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

SQUAREHEAD TECHNOLOGY

MULTI THREADED EXAMPLE

```
1  Data data;
2  bool ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

SQUAREHEAD TECHNOLOGY

MULTI THREADED EXAMPLE

```
1  Data data;
2  bool ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

MULTI THREADED EXAMPLE

```
1  Data data;
2  bool ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

SQUAREHEAD TECHNOLOGY

MULTI THREADED EXAMPLE

```
1  Data data;
2  bool ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

MULTI THREADED EXAMPLE (X86)

```
1  Data data;
2  bool ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

```
1 producer():
2   sub  rsp, 8
3   call initializeData()
4   mov  BYTE PTR ready[rip], 1
5   add  rsp, 8
6   ret
7 consumer():
8   jmp     useData()
```

MULTI THREADED EXAMPLE (X86)

```
1  Data data;
2  bool ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

```
1 producer():
2   sub  rsp, 8
3   call initializeData()
4   mov  BYTE PTR ready[rip], 1
5   add  rsp, 8
6   ret
7 consumer():
8   jmp     useData()
```

MULTI THREADED EXAMPLE (X86)

```
1  Data data;
2  bool ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

```
1 producer():
2   sub  rsp, 8
3   call initializeData()
4   mov  BYTE PTR ready[rip], 1
5   add  rsp, 8
6   ret
7 consumer():
8   jmp     useData()
```

MULTI THREADED EXAMPLE (X86)

```
1  Data data;
2  bool ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

```
1 producer():
2   sub  rsp, 8
3   call initializeData()
4   mov  BYTE PTR ready[rip], 1
5   add  rsp, 8
6   ret
7 consumer():
8   jmp     useData()
```

- Informally:

MULTI THREADED EXAMPLE (X86)

```
1  Data data;
2  bool ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

```
1 producer():
2   sub  rsp, 8
3   call initializeData()
4   mov  BYTE PTR ready[rip], 1
5   add  rsp, 8
6   ret
7 consumer():
8   jmp     useData()
```

- Informally: Always true

SQUAREHEAD TECHNOLOGY

MULTI THREADED EXAMPLE (X86)

```
1  Data data;
2  bool ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

```
1 producer():
2   sub  rsp, 8
3   call initializeData()
4   mov  BYTE PTR ready[rip], 1
5   add  rsp, 8
6   ret
7 consumer():
8   jmp     useData()
```

- Informally: Always true / always false (UB)

MULTI THREADED EXAMPLE (X86)

```
1  Data data;
2  bool ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

```
1 producer():
2   sub  rsp, 8
3   call initializeData()
4   mov  BYTE PTR ready[rip], 1
5   add  rsp, 8
6   ret
7 consumer():
8   jmp     useData()
```

- Informally: Always true / always false (UB) / data race (UB)

MULTI THREADED EXAMPLE (X86)

```
1  Data data;
2  bool ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

```
1 producer():
2   sub  rsp, 8
3   call initializeData()
4   mov  BYTE PTR ready[rip], 1
5   add  rsp, 8
6   ret
7 consumer():
8   jmp     useData()
```

- Informally: Always true / always false (UB) / data race (UB)
- Forward progress: Terminate, I/O, volatile, or sync/atomic

SQUAREHEAD TECHNOLOGY

MULTI THREADED EXAMPLE (NON UB)

```
1  Data data;
2  std::atomic<bool> ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

SQUAREHEAD TECHNOLOGY

MULTI THREADED EXAMPLE (NON UB)

```
1  Data data;
2  std::atomic<bool> ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

SQUAREHEAD TECHNOLOGY

MULTI THREADED EXAMPLE (NON UB X86)

```
1  Data data;
2  std::atomic<bool> ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

```
1  producer():
2    sub     rsp, 8
3    call    initializeData()
4    mov     eax, 1
5    xchg    al, BYTE PTR ready[rip]
6    add     rsp, 8
7    ret
8  consumer():
9  .L5:
10   movzx   eax, BYTE PTR ready[rip]
11   test    al, al
12   je      .L5
13   jmp     useData()
```

SQUAREHEAD TECHNOLOGY

MULTI THREADED EXAMPLE (NON UB X86)

```
1  Data data;
2  std::atomic<bool> ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

```
1  producer():
2    sub     rsp, 8
3    call    initializeData()
4    mov     eax, 1
5    xchg    al, BYTE PTR ready[rip]
6    add     rsp, 8
7    ret
8  consumer():
9  .L5:
10   movzx   eax, BYTE PTR ready[rip]
11   test    al, al
12   je      .L5
13   jmp     useData()
```

SQUAREHEAD TECHNOLOGY

MULTI THREADED EXAMPLE (NON UB X86)

```
1  Data data;
2  std::atomic<bool> ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

```
1  producer():
2    sub     rsp, 8
3    call    initializeData()
4    mov     eax, 1
5    xchg    al, BYTE PTR ready[rip]
6    add     rsp, 8
7    ret
8  consumer():
9  .L5:
10   movzx   eax, BYTE PTR ready[rip]
11   test    al, al
12   je      .L5
13   jmp     useData()
```

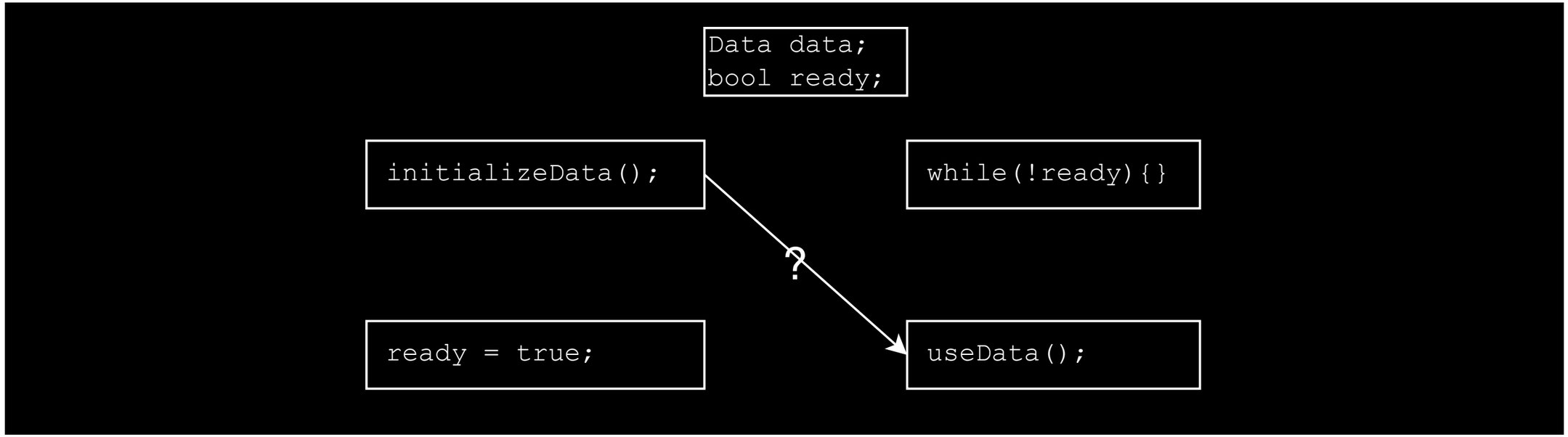
MULTI THREADED EXAMPLE (NON UB X86)

```
1  Data data;
2  std::atomic<bool> ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready = true;
8  }
9
10 void consumer()
11 {
12     while(!ready){}
13     useData();
14 }
```

```
1  producer():
2    sub     rsp, 8
3    call    initializeData()
4    mov     eax, 1
5    xchg    al, BYTE PTR ready[rip]
6    add     rsp, 8
7    ret
8  consumer():
9  .L5:
10   movzx   eax, BYTE PTR ready[rip]
11   test    al, al
12   je      .L5
13   jmp     useData()
```

- Is the data initialized?
- Is the cache up to date?
- Is there a data race?
- Microarchitectural details we don't know about?

- Is the data initialized?
- Is the cache up to date?
- Is there a data race?
- Microarchitectural details we don't know about?



SQUAREHEAD
TECHNOLOGY

DATA RACE

DATA RACE

- Two expression evaluations conflict if **one of them modifies** a memory location and the **other one reads or modifies** the same memory location

DATA RACE

- Two expression evaluations conflict if **one of them modifies** a memory location and the **other one reads or modifies** the same memory location (This is fine)

DATA RACE

- Two expression evaluations conflict if **one of them modifies** a memory location and the **other one reads or modifies** the same memory location (This is fine)
- The execution of a program contains a data race if it contains two potentially concurrent conflicting actions,

DATA RACE

- Two expression evaluations conflict if **one of them modifies** a memory location and the **other one reads or modifies** the same memory location (This is fine)
- The execution of a program contains a data race if it contains two potentially concurrent conflicting actions, at least one of which is not atomic,

DATA RACE

- Two expression evaluations conflict if **one of them modifies** a memory location and the **other one reads or modifies** the same memory location (This is fine)
- The execution of a program contains a data race if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and **neither happens before the other**.

DATA RACE

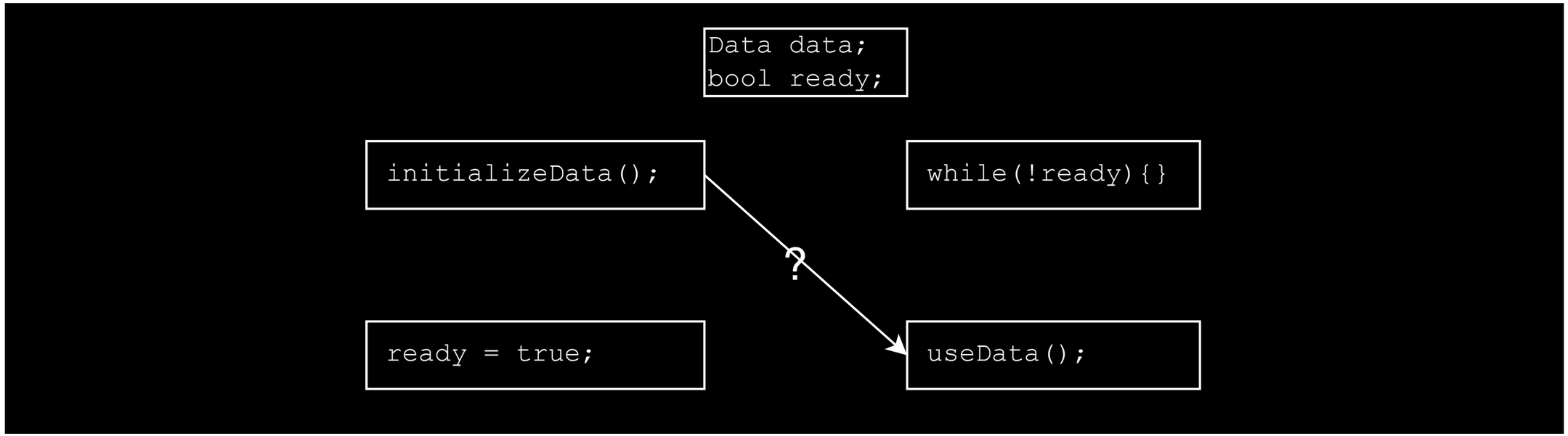
- Two expression evaluations conflict if **one of them modifies** a memory location and the **other one reads or modifies** the same memory location (This is fine)
- The execution of a program contains a data race if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and **neither happens before the other**. Any such data race results in undefined behavior.

DATA RACE

- Two expression evaluations conflict if **one of them modifies** a memory location and the **other one reads or modifies** the same memory location (This is fine)
- The execution of a program contains a data race if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and **neither happens before the other**. Any such data race results in undefined behavior.
- Does the data write **happen before** the read?

SQUAREHEAD TECHNOLOGY

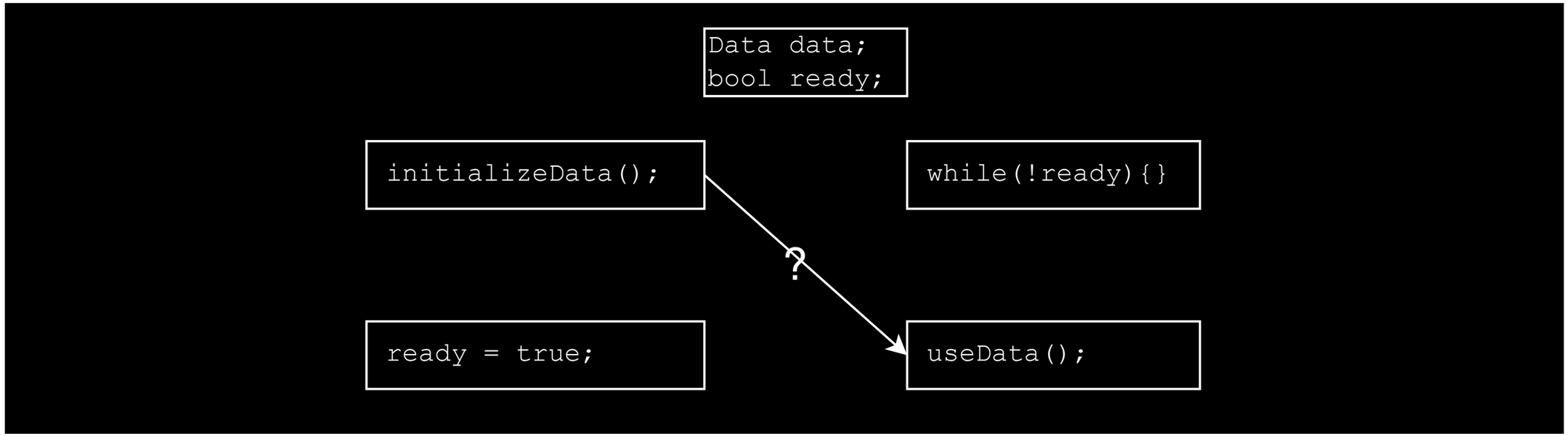
NON ATOMIC



Does `initializeData` **happen before** `useData`?

SQUAREHEAD TECHNOLOGY

NON ATOMIC

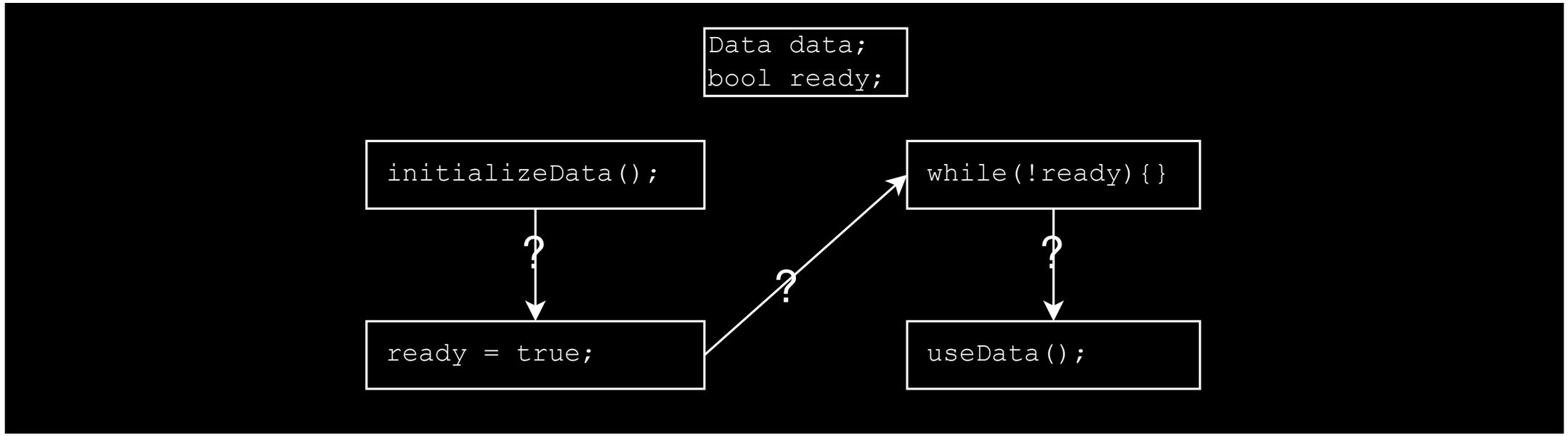


Does `initializeData` **happen before** `useData`?

Is there a *happens-before* relationship here?

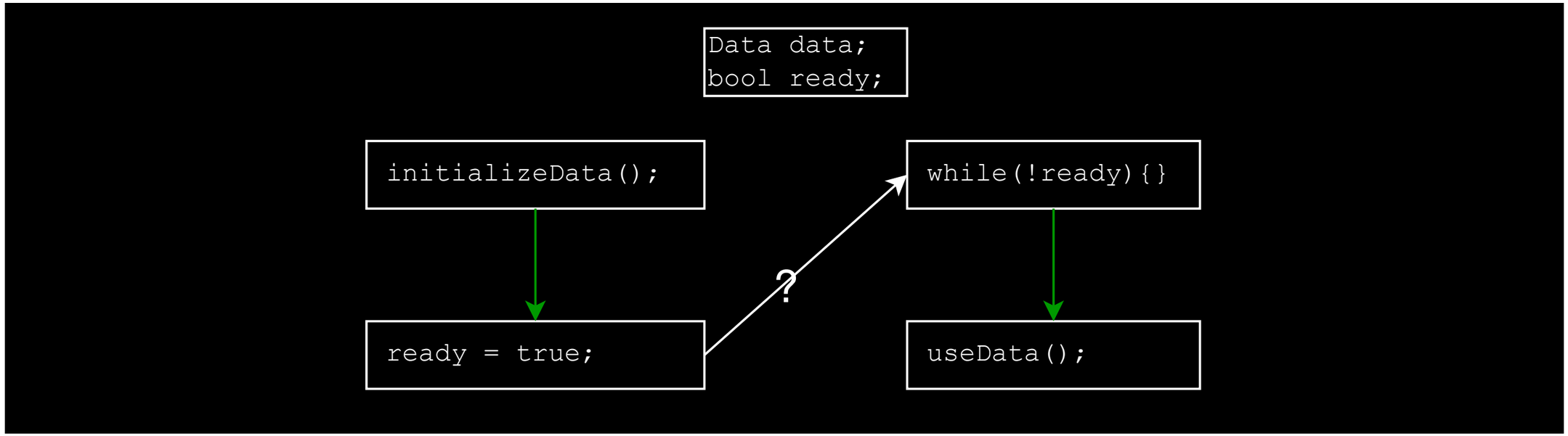
SQUAREHEAD TECHNOLOGY

NON ATOMIC



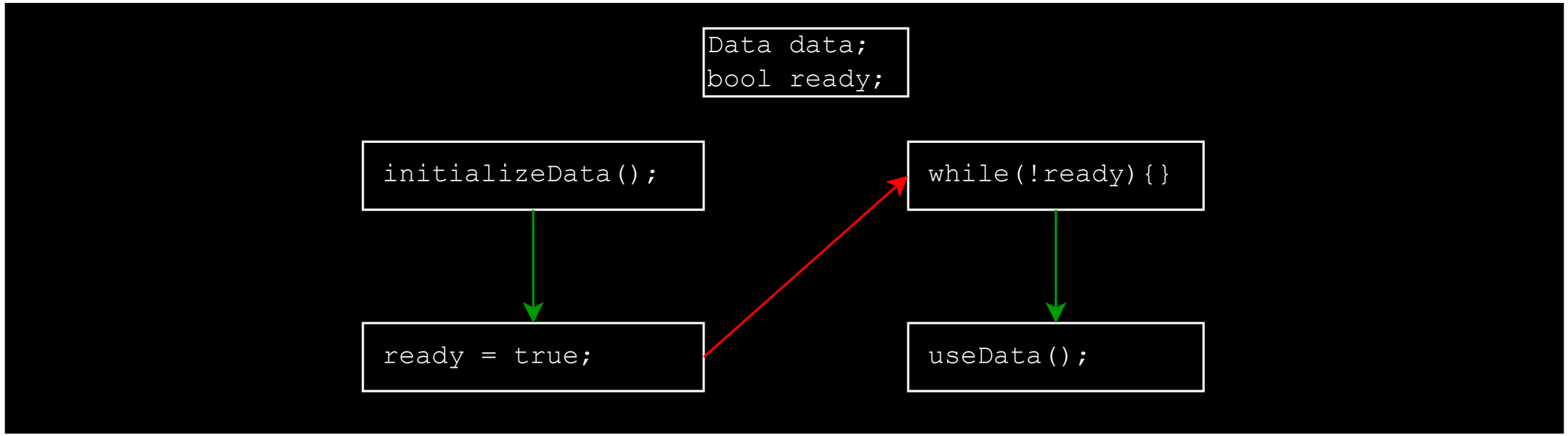
SQUAREHEAD TECHNOLOGY

NON ATOMIC



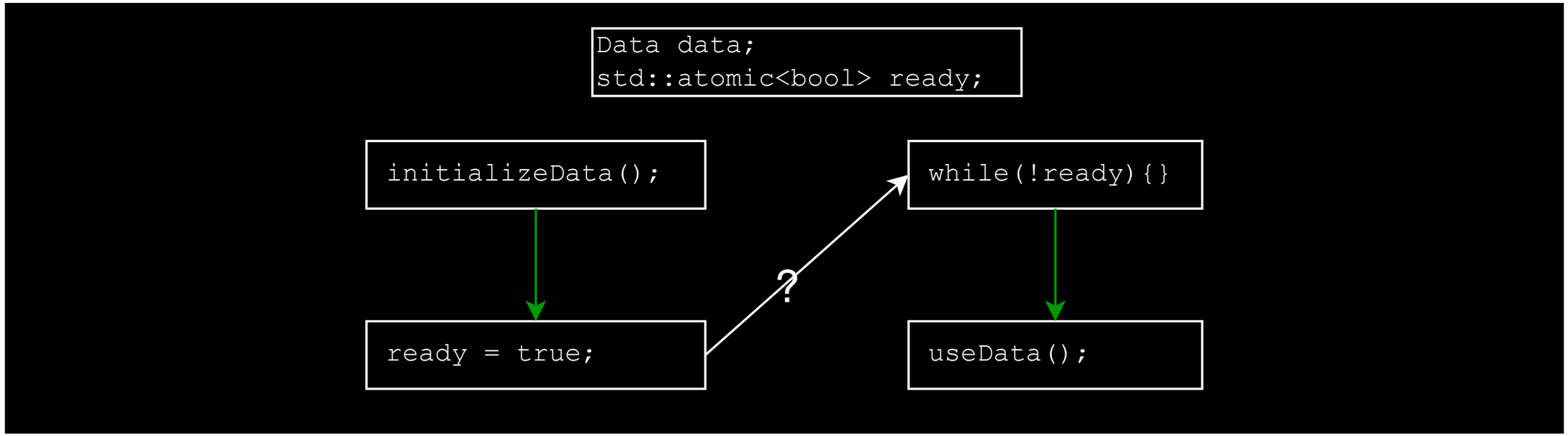
SQUAREHEAD TECHNOLOGY

NON ATOMIC



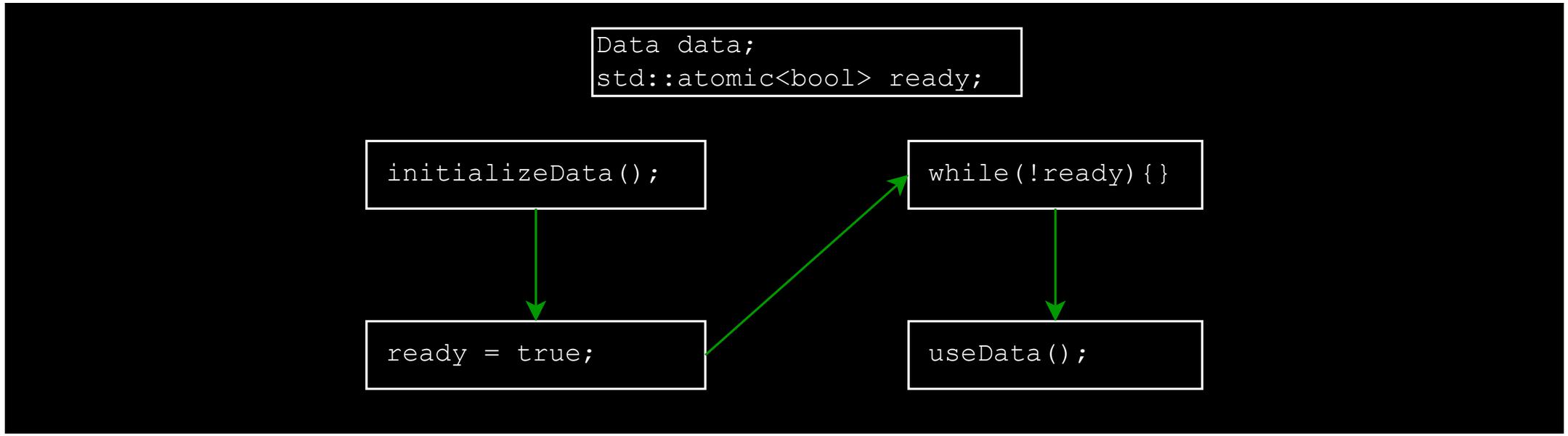
SQUAREHEAD TECHNOLOGY

ATOMIC



SQUAREHEAD TECHNOLOGY

ATOMIC



SQUAREHEAD
TECHNOLOGY

ATOMIC

ATOMIC

- Atomics guarantee no half-written values

ATOMIC

- Atomics guarantee no half-written values
- There is a total order to modifications of each *individual* atomic

ATOMIC

- Atomics guarantee no half-written values
- There is a total order to modifications of each *individual* atomic
- Otherwise, *it depends*

SQUAREHEAD TECHNOLOGY

MEMORY ORDER

```
Data data;
std::atomic<bool> ready{false};

void producer()
{
    initializeData();
    ready = true;
}

void consumer()
{
    while(!ready){}
    useData();
}
```

MEMORY ORDER

```
1  Data data;
2  std::atomic<bool> ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready.store(true, std::memory_order_seq_cst);
8  }
9
10 void consumer()
11 {
12     while(!ready.load(std::memory_order_seq_cst)){}
13     useData();
14 }
```

SQUAREHEAD TECHNOLOGY

MEMORY ORDER

```
1  Data data;
2  std::atomic<bool> ready{false};
3
4  void producer()
5  {
6      initializeData();
7      ready.store(true, std::memory_order_seq_cst);
8  }
9
10 void consumer()
11 {
12     while(!ready.load(std::memory_order_seq_cst)){}
13     useData();
14 }
```

SQUAREHEAD TECHNOLOGY

MEMORY ORDER

- Sequentially consistent:

MEMORY ORDER

- Sequentially consistent:
  - Establishes happens-before across threads

MEMORY ORDER

- Sequentially consistent:
  - Establishes happens-before across threads
  - Single total order for all seq_cst operations

MEMORY ORDER

- Sequentially consistent:
  - Establishes happens-before across threads
  - Single total order for all seq_cst operations
  - Stricter than necessary about reordering

MEMORY ORDER

- Sequentially consistent:
  - Establishes happens-before across threads
  - Single total order for all seq_cst operations
  - Stricter than necessary about reordering
- Relaxed:

MEMORY ORDER

- Sequentially consistent:
  - Establishes happens-before across threads
  - Single total order for all seq_cst operations
  - Stricter than necessary about reordering
- Relaxed:
  - No happens-before across threads

MEMORY ORDER

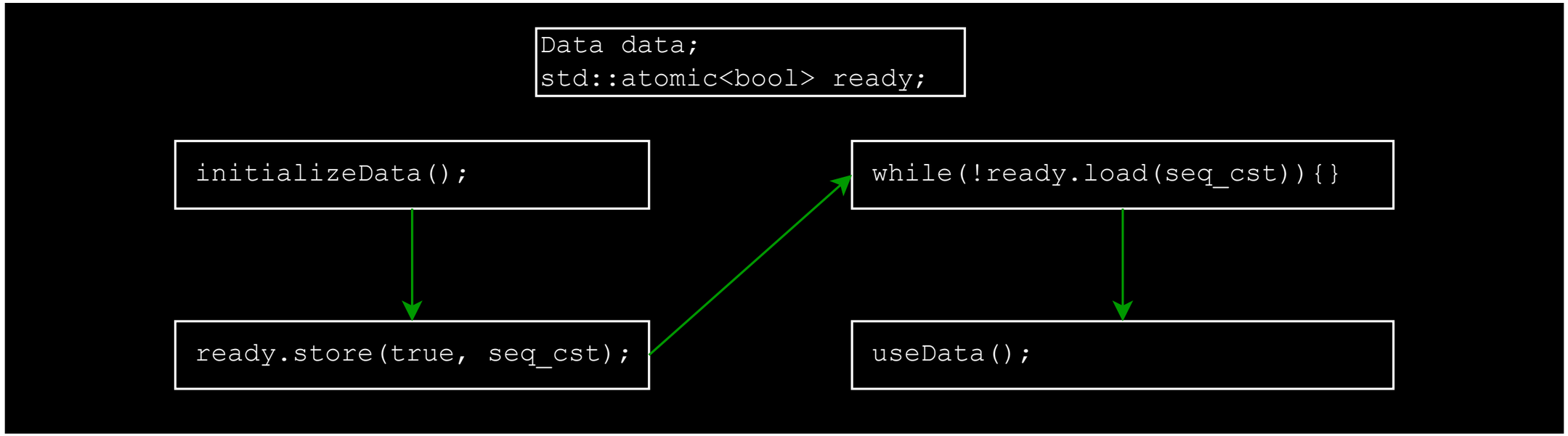
- Sequentially consistent:
  - Establishes happens-before across threads
  - Single total order for all seq_cst operations
  - Stricter than necessary about reordering
- Relaxed:
  - No happens-before across threads
  - One total modification order *per atomic*

MEMORY ORDER

- Sequentially consistent:
  - Establishes happens-before across threads
  - Single total order for all seq_cst operations
  - Stricter than necessary about reordering
- Relaxed:
  - No happens-before across threads
  - One total modification order *per atomic*
- Avoid half-written writes

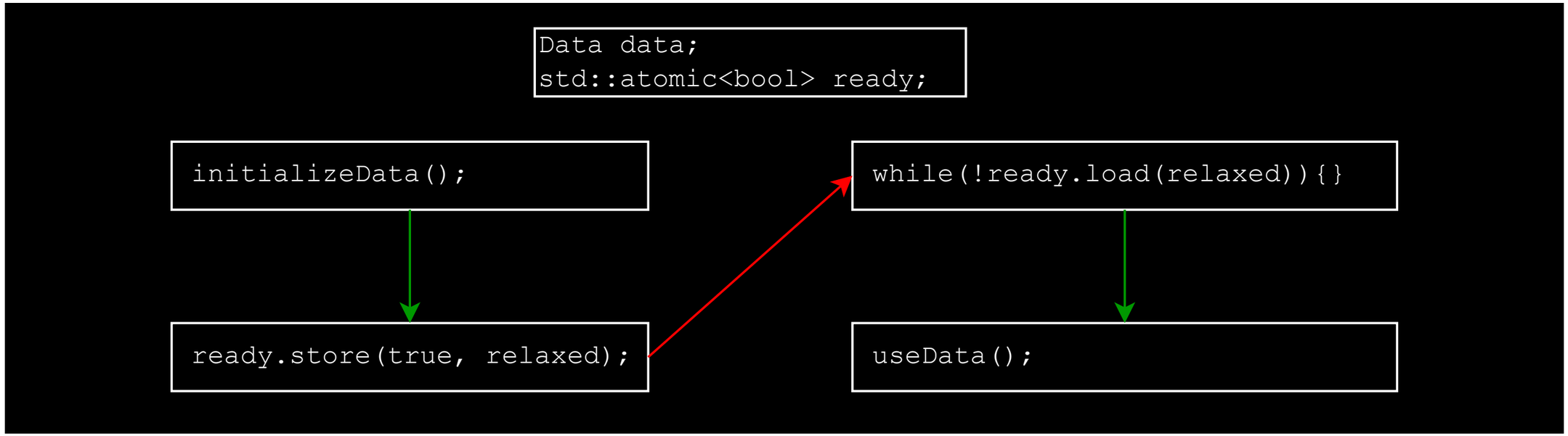
SQUAREHEAD TECHNOLOGY

SEQUENTIALLY CONSISTENT



SQUAREHEAD TECHNOLOGY

RELAXED

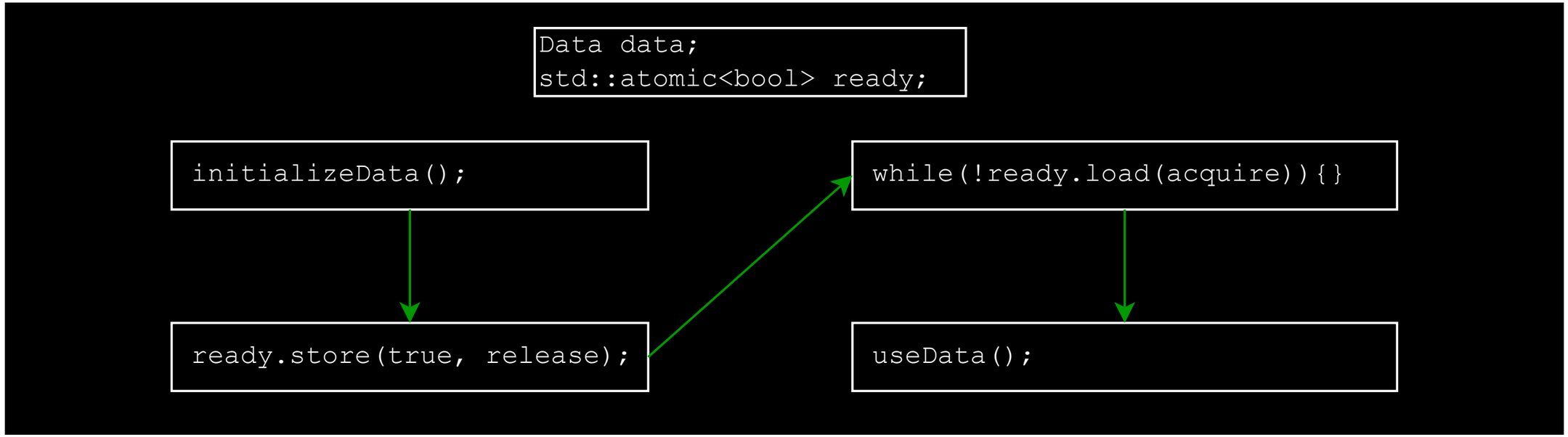


ACQUIRE / RELEASE

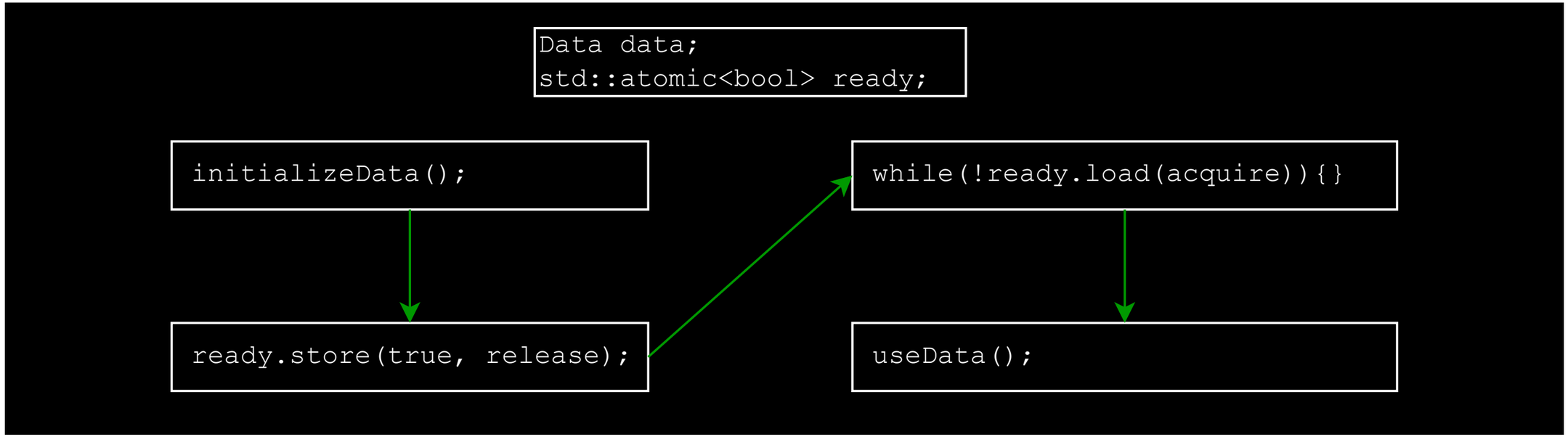
If an acquire-load observes the value of a release-store, the release-store happens before the acquire-load

SQUAREHEAD TECHNOLOGY

RELEASE/ACQUIRE



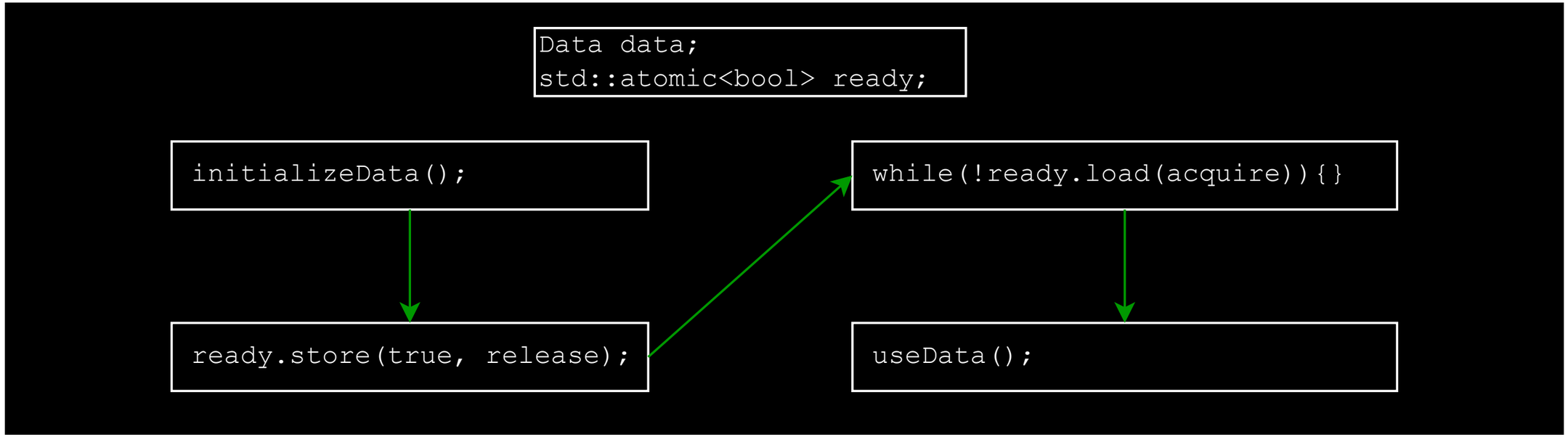
RELEASE/ACQUIRE



- Store happens-before load

SQUAREHEAD TECHNOLOGY

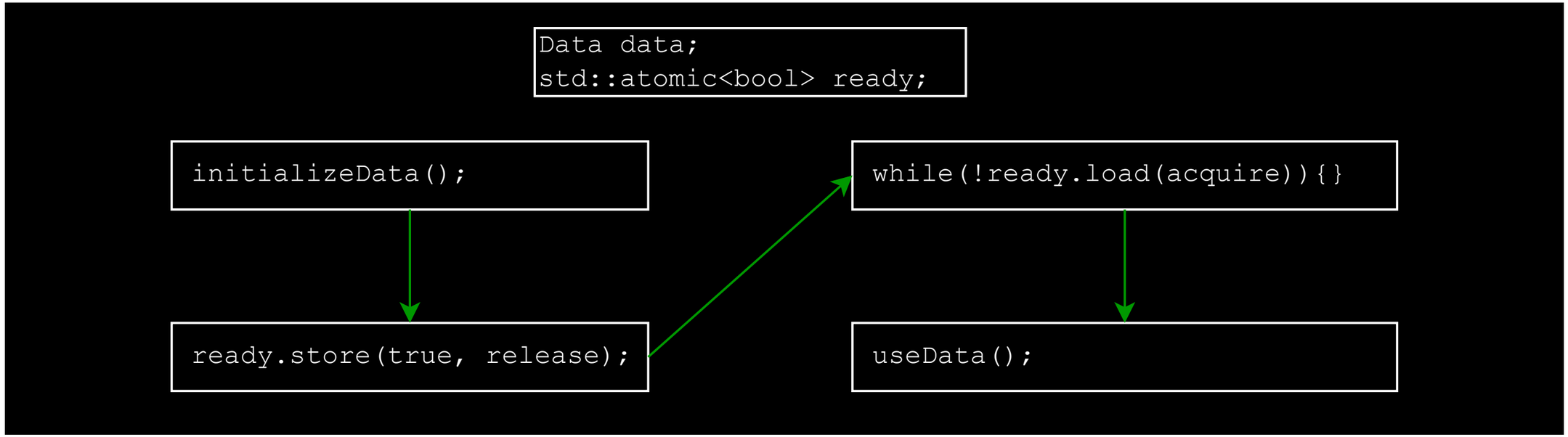
RELEASE/ACQUIRE



- Store happens-before load
- Can't reorder memory ops after the release-store

SQUAREHEAD TECHNOLOGY

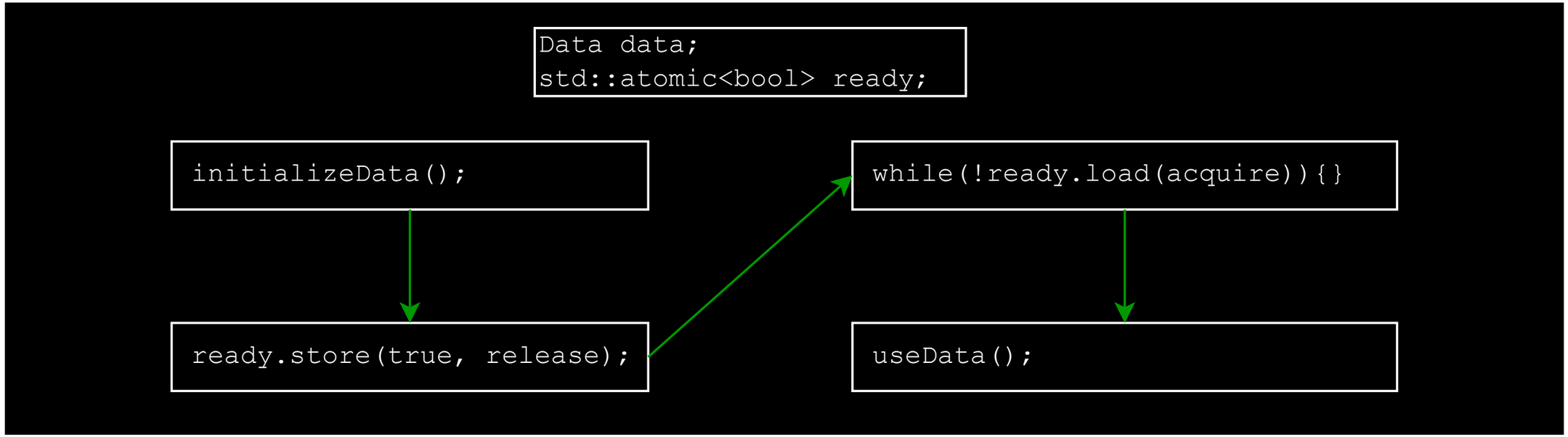
RELEASE/ACQUIRE



- Store happens-before load
- Can't reorder memory ops after the release-store
- Can't reorder memory ops before the acquire-load

SQUAREHEAD TECHNOLOGY

RELEASE/ACQUIRE



- Store happens-before load
- Can't reorder memory ops after the release-store
- Can't reorder memory ops before the acquire-load
- But opposite is ok

MULTI THREADED EXAMPLE (X86)

```
Data data;
std::atomic<bool> ready{false};

void producer()
{
    initializeData();
    ready.store(true, std::memory_order_releas
}

void consumer()
{
    while(!ready.load(std::memory_order_acquir
    useData();
}
```

MULTI THREADED EXAMPLE (X86)

```
Data data;
std::atomic<bool> ready{false};

void producer()
{
    initializeData();
    ready.store(true, std::memory_order_releas
}

void consumer()
{
    while(!ready.load(std::memory_order_acquir
    useData();
}
```

```
 1 producer():
 2   sub     rsp, 8
 3   call    initializeData()
 4   mov     BYTE PTR ready[rip], 1
 5   add     rsp, 8
 6   ret
 7 consumer():
 8 .L5:
 9   movzx   eax, BYTE PTR ready[rip
10   test    al, al
11   je      .L5
12   jmp     useData()
```

SQUAREHEAD TECHNOLOGY

MULTI THREADED EXAMPLE (X86)

```cpp
Data data;
std::atomic<bool> ready{false};

void producer()
{
    initializeData();
    ready.store(true, std::memory_order_releas
}

void consumer()
{
    while(!ready.load(std::memory_order_acquir
    useData();
}
```

```
 1 producer():
 2   sub     rsp, 8
 3   call    initializeData()
 4   mov     BYTE PTR ready[rip], 1
 5   add     rsp, 8
 6   ret
 7 consumer():
 8 .L5:
 9   movzx   eax, BYTE PTR ready[rip
10   test    al, al
11   je      .L5
12   jmp     useData()
```

Also, `memory_order::`

- `acq_rel` (read-modify-write ops)
- `consume` (don't use)

How does the compiler know what is safe and not?

SQUAREHEAD
TECHNOLOGY

CPU MEMORY MODEL

SQUAREHEAD TECHNOLOGY

WHAT DO WE MEAN BY (RE)ORDERING?

WHAT DO WE MEAN BY (RE)ORDERING?

- Out of order instruction execution?

WHAT DO WE MEAN BY (RE)ORDERING?

- Out of order instruction execution?
- Out of order micro-op execution?

WHAT DO WE MEAN BY (RE)ORDERING?

- Out of order instruction execution?
- Out of order micro-op execution?
- Pipeline?

WHAT DO WE MEAN BY (RE)ORDERING?

- Out of order instruction execution?
- Out of order micro-op execution?
- Pipeline?
- Forwarding networks?

WHAT DO WE MEAN BY (RE)ORDERING?

- Out of order instruction execution?
- Out of order micro-op execution?
- Pipeline?
- Forwarding networks?
- Store buffers?

WHAT DO WE MEAN BY (RE)ORDERING?

- Out of order instruction execution?
- Out of order micro-op execution?
- Pipeline?
- Forwarding networks?
- Store buffers?
- Store/load coalescing?

WHAT DO WE MEAN BY (RE)ORDERING?

- Out of order instruction execution?
- Out of order micro-op execution?
- Pipeline?
- Forwarding networks?
- Store buffers?
- Store/load coalescing?
- Memory system?

WHAT DO WE MEAN BY (RE)ORDERING?

- Out of order instruction execution?
- Out of order micro-op execution?
- Pipeline?
- Forwarding networks?
- Store buffers?
- Store/load coalescing?
- Memory system?
- Caches?

SQUAREHEAD TECHNOLOGY

MEMORY (CONSISTENCY) MODEL

MEMORY (CONSISTENCY) MODEL

- Different way of thinking than C++ memory model!

MEMORY (CONSISTENCY) MODEL

- Different way of thinking than C++ memory model!
- Imagine a single global order of all memory operations

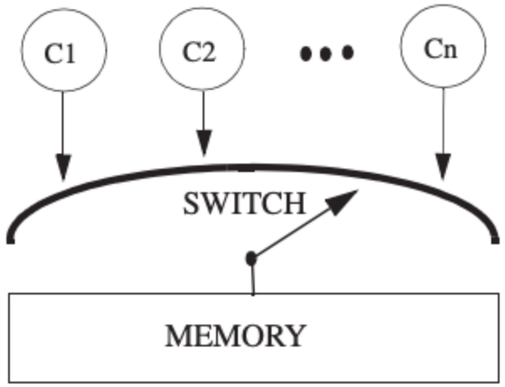
MEMORY (CONSISTENCY) MODEL

- Different way of thinking than C++ memory model!
- Imagine a single global order of all memory operations
- What are the allowed orders

SQUAREHEAD TECHNOLOGY

MEMORY (CONSISTENCY) MODEL

- Different way of thinking than C++ memory model!
- Imagine a single global order of all memory operations
- What are the allowed orders



Sarin, Hill, Wood: A Primer on Memory Consistency and Cache Coherence

SQUAREHEAD TECHNOLOGY

PROGRAM ORDER

Producer thread

```
mov reg_0 42
store *data reg_0
mov reg_1 1
store *ready reg_1
```

PROGRAM ORDER

Producer thread

```
mov reg_0 42
store *data reg_0
mov reg_1 1
store *ready reg_1
```

Consumer thread

```
.wait
load reg_0 *ready
jz reg_0 .wait
load reg_1 *data
(use reg_1)
```

PROGRAM ORDER

Producer thread

```
store *data reg_0
store *ready reg_1
```

Consumer thread

```
load reg_0 *ready
load reg_1 *data
```

PROGRAM ORDER

Producer thread	Consumer thread
`store *data reg_0`	`load reg_0 *ready`
`store *ready reg_1`	`load reg_1 *data`

What is a valid order here?

PROGRAM ORDER

Producer thread	Consumer thread
`store *data reg_0`	`load reg_0 *ready`
`store *ready reg_1`	`load reg_1 *data`

What is a valid order here?

- In the (imaginary) global, total memory order,

PROGRAM ORDER

Producer thread	Consumer thread
`store *data reg_0`	`load reg_0 *ready`
`store *ready reg_1`	`load reg_1 *data`

What is a valid order here?

- In the (imaginary) global, total memory order,
- imagine each thread takes turns (as-if rule).

PROGRAM ORDER

Producer thread	Consumer thread
`store *data reg_0`	`load reg_0 *ready`
`store *ready reg_1`	`load reg_1 *data`

What is a valid order here?

- In the (imaginary) global, total memory order,
- imagine each thread takes turns (as-if rule).
- How much of program order is preserved?

MEMORY ORDER

Producer thread

```
store *data reg_0
store *ready reg_1
```

Memory order

```
store *data reg_0
store *ready reg_1
load reg_0 *ready
load reg_1 *data
```

Consumer thread

```
load reg_0 *ready
load reg_1 *data
```

SQUAREHEAD TECHNOLOGY

MEMORY ORDER

Producer thread	Memory order	Consumer thread
`store *data reg_0`	`store *data reg_0`	`load reg_0 *ready`
`store *ready reg_1`	`load reg_0 *ready`	`load reg_1 *data`
	`load reg_0 *ready`	
	`store *ready reg_1`	
	`load reg_0 *ready`	
	`load reg_1 *data`	

SQUAREHEAD TECHNOLOGY

MEMORY ORDER

Producer thread	Memory order	Consumer thread
`store *data reg_0`	`load reg_0 *ready`	`load reg_0 *ready`
`store *ready reg_1`	`load reg_0 *ready`	`load reg_1 *data`
	`store *ready reg_1`	
	`load reg_0 *ready`	
	`load reg_1 *data`	
	`store *data reg_0`	

SQUAREHEAD TECHNOLOGY

MEMORY ORDER

Producer thread	Memory order	Consumer thread
`store *data reg_0`	`store *data reg_0`	`load reg_0 *ready`
`store *ready reg_1`	`load reg_0 *ready`	`load reg_1 *data`
	`load reg_0 *ready`	
	`store *ready reg_1`	
	`load reg_0 *ready`	
	`load reg_1 *data`	

- Sequential consistency

SQUAREHEAD TECHNOLOGY

MEMORY ORDER

Producer thread

```
store *data reg_0
store *ready reg_1
```

Memory order

```
store *data reg_0
load reg_0 *ready
load reg_0 *ready
store *ready reg_1
load reg_0 *ready
load reg_1 *data
```

Consumer thread

```
load reg_0 *ready
load reg_1 *data
```

- Sequential consistency
- All of program order is preserved

SQUAREHEAD TECHNOLOGY

MEMORY ORDER

Producer thread

```
store *data reg_0
store *ready reg_1
```

Memory order

```
store *data reg_0
load reg_0 *ready
load reg_0 *ready
store *ready reg_1
load reg_0 *ready
load reg_1 *data
```

Consumer thread

```
load reg_0 *ready
load reg_1 *data
```

- Sequential consistency
- All of program order is preserved
- Easy to reason about

SQUAREHEAD TECHNOLOGY

MEMORY ORDER

Producer thread	Memory order	Consumer thread
`store *data reg_0`	`store *data reg_0`	`load reg_0 *ready`
`store *ready reg_1`	`load reg_0 *ready`	`load reg_1 *data`
	`load reg_0 *ready`	
	`store *ready reg_1`	
	`load reg_0 *ready`	
	`load reg_1 *data`	

- Sequential consistency
- All of program order is preserved
- Easy to reason about
- Less possibilities for optimization ("no-one" does this)

SEQUENTIAL CONSISTENCY

Preserves

- Load → Load
- Store → Store
- Load → Store
- Store → Load

TOTAL STORE ORDERING (TSO) (~X86)

Preserves

- Load → Load
- Store → Store
- Load → Store
- ~~Store → Load~~

TOTAL STORE ORDERING (TSO) (~X86)

- Motivated by store buffers

TOTAL STORE ORDERING (TSO) (~X86)

- Motivated by store buffers
- Stores are buffered on the core into a store buffer

TOTAL STORE ORDERING (TSO) (~X86)

- Motivated by store buffers
- Stores are buffered on the core into a store buffer
- A later load might take its value before the buffer is drained to cache

TSO EXAMPLE

Thread 1	Thread 2
`store x 1`	`store y 1`
`load y`	`load x`

TSO EXAMPLE

Thread 1	Thread 2
`store x 1`	`store y 1`
`load y`	`load x`

Can `x` *and* `y` be `0`?

SQUAREHEAD TECHNOLOGY

TSO EXAMPLE

Thread 1	Memory order	Thread 2
`store x 1`	`load y`	`store y 1`
`load y`	`store y 1`	`load x`
	`load x`	
	`store x 1`	

SQUAREHEAD TECHNOLOGY

TSO EXAMPLE

Thread 1	Memory order	Thread 2
`store x 1`	`load y`	`store y 1`
`load y`	`store y 1`	`load x`
	`load x`	
	`store x 1`	

`x == 0, y == 0`

Fine on TSO, not on SC!

CLARIFICATION

Thread 1

```
store x 1
```

```
load x
```

CLARIFICATION

Thread 1

```
store x 1
```

```
load x
```

Can `store x` be ordered after `load x`?

SQUAREHEAD TECHNOLOGY

CLARIFICATION

Thread 1	Memory order
`store x 1`	`load x`
`load x`	`store x 1`

SQUAREHEAD TECHNOLOGY

CLARIFICATION

Thread 1	Memory order
`store x 1`	`load x`
`load x`	`store x 1`

Yes! But.

SQUAREHEAD TECHNOLOGY

CLARIFICATION

Thread 1	Memory order
`store x 1`	`load x`
`load x`	`store x 1`

Yes! But.

- Memory order: The order that memory operations go to a hypothetical serialized memory

SQUAREHEAD TECHNOLOGY

CLARIFICATION

Thread 1	Memory order
`store x 1`	`load x`
`load x`	`store x 1`

Yes! But.

- Memory order: The order that memory operations go to a hypothetical serialized memory
- Value of a load follows program order, otherwise memory order

CLARIFICATION

Thread 1	Memory order
`store x 1`	`load x`
`load x`	`store x 1`

Yes! But.

- Memory order: The order that memory operations go to a hypothetical serialized memory
- Value of a load follows program order, otherwise memory order
- `load x` picks `1` from the core's store buffer

CLARIFICATION

Thread 1	Memory order
`store x 1`	`load x`
`load x`	`store x 1`

Yes! But.

- Memory order: The order that memory operations go to a hypothetical serialized memory
- Value of a load follows program order, otherwise memory order
- `load x` picks `1` from the core's store buffer
- `load x` never even goes to cache

SQUAREHEAD TECHNOLOGY

CLARIFICATION

Thread 1	Memory order
`store x 1`	`load x`
`load x`	`store x 1`

Yes! But.

- Memory order: The order that memory operations go to a hypothetical serialized memory
- Value of a load follows program order, otherwise memory order
- `load x` picks `1` from the core's store buffer
- `load x` never even goes to cache
- Just a model

MULTI THREADED EXAMPLE (X86/TSO)

```
Data data;
std::atomic<bool> ready{false};

void producer()
{
    initializeData();
    ready.store(true, std::memory_order_releas
}

void consumer()
{
    while(!ready.load(std::memory_order_acquir
    useData();
}
```

```
 1 producer():
 2   sub     rsp, 8
 3   call    initializeData()
 4   mov     BYTE PTR ready[rip], 1
 5   add     rsp, 8
 6   ret
 7 consumer():
 8 .L5:
 9   movzx   eax, BYTE PTR ready[rip
10   test    al, al
11   je      .L5
12   jmp     useData()
```

SQUAREHEAD TECHNOLOGY

MULTI THREADED EXAMPLE (X86/TSO)

```cpp
Data data;
std::atomic<bool> ready{false};

void producer()
{
    initializeData();
    ready.store(true, std::memory_order_releas
}

void consumer()
{
    while(!ready.load(std::memory_order_acquir
    useData();
}
```

```
 1 producer():
 2   sub     rsp, 8
 3   call    initializeData()
 4   mov     BYTE PTR ready[rip], 1
 5   add     rsp, 8
 6   ret
 7 consumer():
 8 .L5:
 9   movzx   eax, BYTE PTR ready[rip
10   test    al, al
11   je      .L5
12   jmp     useData()
```

RELAXED MEMORY MODELS (ARM, RISC-V)

- Do we really need to keep order of *all* stores

RELAXED MEMORY MODELS (ARM, RISC-V)

- Do we really need to keep order of *all* stores and *all* loads

RELAXED MEMORY MODELS (ARM, RISC-V)

- Do we really need to keep order of *all* stores and *all* loads and *all loads before stores*?

RELAXED MEMORY MODELS (ARM, RISC-V)

- Do we really need to keep order of *all* stores and *all* loads and *all loads before stores*?
- What if want non-FIFO store buffers?

RELAXED MEMORY MODELS (ARM, RISC-V)

- Do we really need to keep order of *all* stores and *all* loads and *all loads before stores*?
- What if want non-FIFO store buffers?
- What if we want coalescing of stores/loads?

RELAXED MEMORY MODELS (ARM, RISC-V)

- Do we really need to keep order of *all* stores and *all* loads and *all loads before stores*?
- What if want non-FIFO store buffers?
- What if we want coalescing of stores/loads?
- What if we want fancy speculation and prediction?

RELAXED MEMORY MODELS (ARM, RISC-V)

- Do we really need to keep order of *all* stores and *all* loads and *all loads before stores*?
- What if want non-FIFO store buffers?
- What if we want coalescing of stores/loads?
- What if we want fancy speculation and prediction?
- What if we want other optimizations in the memory system / cache?

SQUAREHEAD TECHNOLOGY

RELAXED MEMORY MODELS

Producer thread

```
store *data0 reg_0
store *data1 reg_1
store *ready reg_2
```

Consumer thread

```
load reg_0 *ready
load reg_1 *data0
load reg_2 *data1
```

SQUAREHEAD TECHNOLOGY

RELAXED MEMORY MODELS

Producer thread

```
store *data0 reg_0
store *data1 reg_1
store *ready reg_2
```

Memory order

```
store *data1 reg_1
store *data0 reg_0
store *ready reg_2
load reg_0 *ready
load reg_2 *data1
load reg_1 *data0
```

Consumer thread

```
load reg_0 *ready
load reg_1 *data0
load reg_2 *data1
```

RELAXED MEMORY MODELS

- As a programmer (with an interest in performance), I want to

RELAXED MEMORY MODELS

- As a programmer (with an interest in performance), I want to
  - Give the architecture a lot of freedom to optimize

RELAXED MEMORY MODELS

- As a programmer (with an interest in performance), I want to
  - Give the architecture a lot of freedom to optimize
  - Tell it which memory operations I care about

RELAXED MEMORY MODELS

- As a programmer (with an interest in performance), I want to
  - Give the architecture a lot of freedom to optimize
  - Tell it which memory operations I care about
  - At least a few safe defaults

RELAXED MEMORY MODELS

- As a programmer (with an interest in performance), I want to
  - Give the architecture a lot of freedom to optimize
  - Tell it which memory operations I care about
  - At least a few safe defaults
  - A memory model to reason about

RELAXED MEMORY MODELS: RISC-V

SAFE DEFAULTS: SAME ADDRESS STORE → STORE

Producer thread

```
store *data0 reg_0
store *data0 reg_1
```

Memory order?

```
store *data0 reg_1
store *data0 reg_0
```

SQUAREHEAD TECHNOLOGY

RELAXED MEMORY MODELS: RISC-V

SAFE DEFAULTS: DATA DEPENDENCY

Producer thread

```
load reg_0 *data0
store *data1 reg_0
```

Memory order?

```
store *data1 reg_0
load reg_0 *data0
```

SQUAREHEAD TECHNOLOGY

RELAXED MEMORY MODELS: RISC-V

SAFE DEFAULTS: ADDRESS DEPENDENCY

Producer thread

```
load reg_0 pointer
store *reg_0 2
```

Memory order?

```
store *reg_0 2
load reg_0 pointer
```

RELAXED MEMORY MODELS: RISC-V

FENCES

Producer thread

```
store *data0 reg_0
store *data1 reg_1
store *ready reg_2
```

Memory order

```
store *data1 reg_1
store *data0 reg_0
store *ready reg_2
load reg_0 *ready
load reg_2 *data1
load reg_1 *data0
```

Consumer thread

```
load reg_0 *ready
load reg_1 *data0
load reg_2 *data1
```

SQUAREHEAD TECHNOLOGY

RELAXED MEMORY MODELS: RISC-V

FENCES

Producer thread

```
store *data0 reg_0
store *data1 reg_1
fence w, w
store *ready reg_2
```

Memory order

```
store *data1 reg_1
store *data0 reg_0
store *ready reg_2
load reg_0 *ready
load reg_2 *data1
load reg_1 *data0
```

Consumer thread

```
load reg_0 *ready
fence r, r
load reg_1 *data0
load reg_2 *data1
```

SQUAREHEAD TECHNOLOGY

RELAXED MEMORY MODELS: RISC-V

RELAXED MEMORY MODELS: RISC-V

- RISC-V allows all sorts of reordering

RELAXED MEMORY MODELS: RISC-V

- RISC-V allows all sorts of reordering
- Except 13 specific rules, e.g.

RELAXED MEMORY MODELS: RISC-V

- RISC-V allows all sorts of reordering
- Except 13 specific rules, e.g.
  - Don't reorder past a store to overlapping address

RELAXED MEMORY MODELS: RISC-V

- RISC-V allows all sorts of reordering
- Except 13 specific rules, e.g.
  - Don't reorder past a store to overlapping address
  - Don't reorder across data/control/address dependency

RELAXED MEMORY MODELS: RISC-V

- RISC-V allows all sorts of reordering
- Except 13 specific rules, e.g.
  - Don't reorder past a store to overlapping address
  - Don't reorder across data/control/address dependency
  - Don't reorder across fences (depending on type of fence)

RELAXED MEMORY MODELS: RISC-V

- RISC-V allows all sorts of reordering
- Except 13 specific rules, e.g.
  - Don't reorder past a store to overlapping address
  - Don't reorder across data/control/address dependency
  - Don't reorder across fences (depending on type of fence)
  - Don't reorder across AMOs with acquire/release semantics

RELAXED MEMORY MODELS: RISC-V

- RISC-V allows all sorts of reordering
- Except 13 specific rules, e.g.
  - Don't reorder past a store to overlapping address
  - Don't reorder across data/control/address dependency
  - Don't reorder across fences (depending on type of fence)
  - Don't reorder across AMOs with acquire/release semantics
  - And more, see "Preserved Program Order" in RISC-V ISA manual

SQUAREHEAD
TECHNOLOGY

Language memory model

SQUAREHEAD TECHNOLOGY

Language memory model

↓

Architecture memory model

↓

Language memory model

↓

Architecture memory model

↓

Microarchitecture (pipeline, store buffers)

Language memory model

↓

Architecture memory model

↓

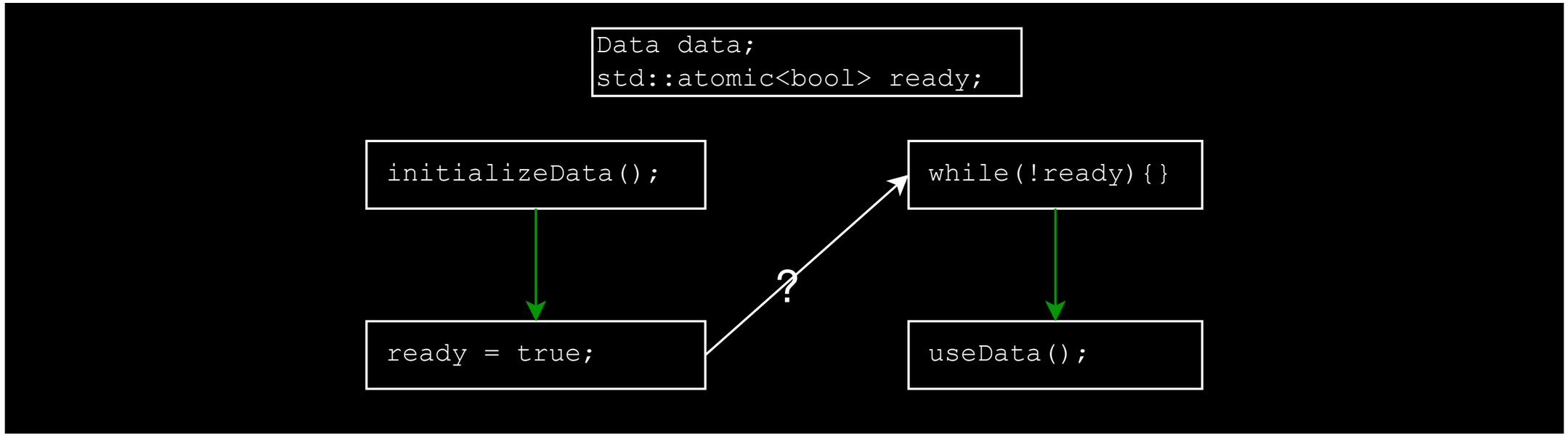
Microarchitecture (pipeline, store buffers) and cache coherence

SQUAREHEAD
TECHNOLOGY

REMEMBER

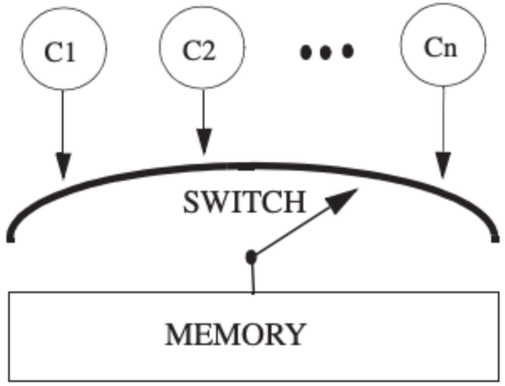
SQUAREHEAD TECHNOLOGY

C++ MEMORY MODEL



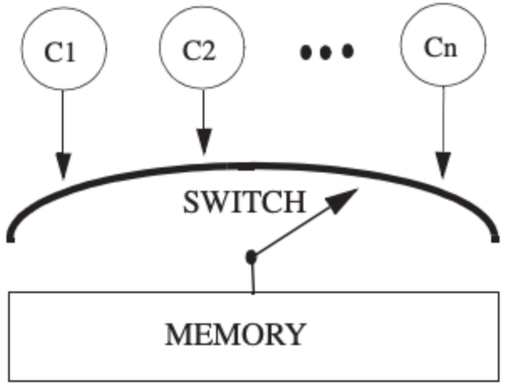
SQUAREHEAD TECHNOLOGY

CPU MEMORY MODEL



SQUAREHEAD TECHNOLOGY

CPU MEMORY MODEL



Producer thread

```
store *data reg_0
store *ready reg_1
```

Memory order

```
store *data reg_0
store *ready reg_1
load reg_0 *ready
load reg_1 *data
```

Consumer thread

```
load reg_0 *ready
load reg_1 *data
```

SQUAREHEAD
TECHNOLOGY

BONUS!

DON'T TRUST THE ASSEMBLY

SQUAREHEAD TECHNOLOGY

THE TWO MEMORY MODELS

ANDERS SCHAU KNATTEN

Squarehead / CppQuiz.org

MEETING C++ 2025