Concurrency

CMSC 240

All examples borrowed/modified from

C++ Crash Course by Josh Lospinoso

No Starch Press

Concurrency vs Parallelism

- Concurrency: Making progress on more than one task at the same time
  - Note this does not mean that any two tasks are being worked on at the exact same time
    - E.g., context switch
- Parallelism: Two or more actions executing simultaneously
  - Requires multiple processing units

Thanks Madhaven Nagarajan:
https://medium.com/@itIsMadhavan/concurrency-vs-parallelism-a-brief-review-b337c8dac350

Concurrency vs Parallelism

- From Art of Concurrency (Clay Breshears): A system is said to be *concurrent* if it can support two or more actions *in progress* at the same time. A system is said to be *parallel* if it can support two or more actions executing simultaneously.
  - term *in progress* is key here

Concurrency vs Parallelism

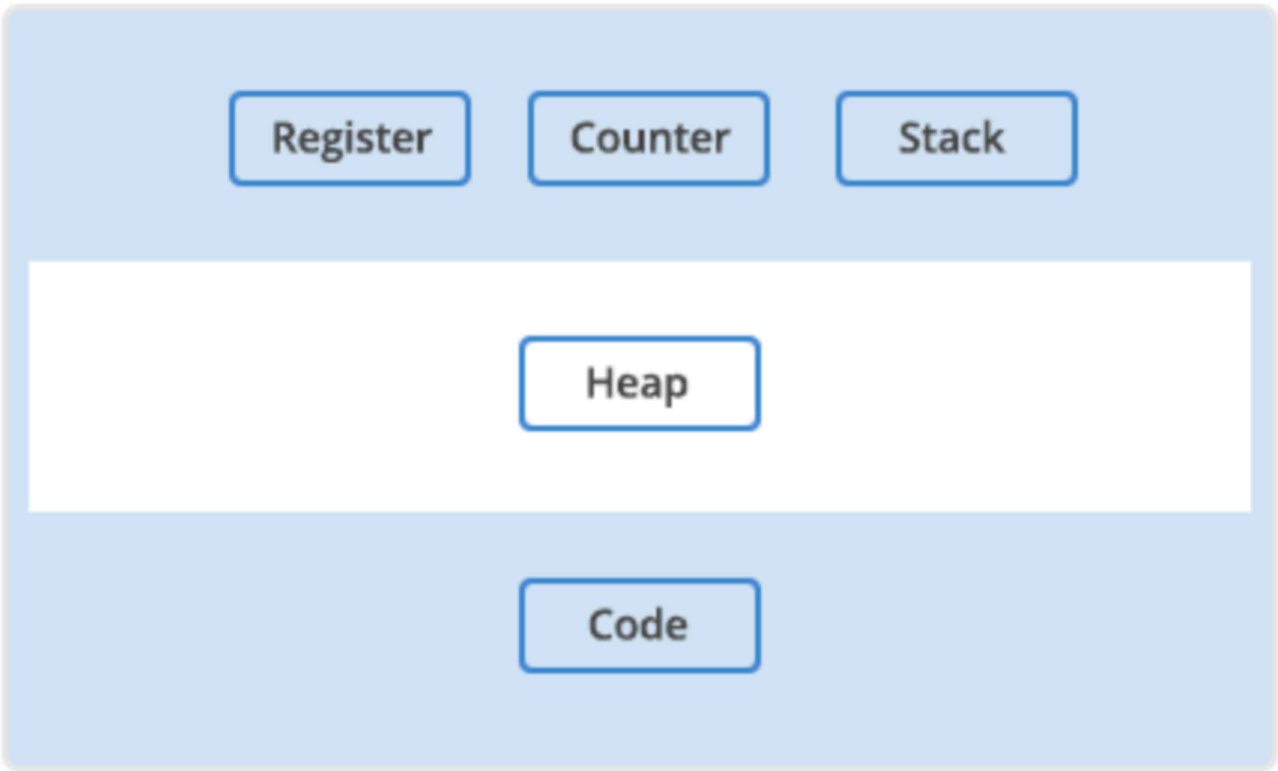
- Concurrency is about **dealing with lots of things** at once. Parallelism is about **doing lots of things** at once.
- Application can be concurrent but not parallel
- Application can be parallel but not concurrent (e.g., single task whose parts are farmed to multiple processors)
  - So you don’t need multiple tasks to have parallelism

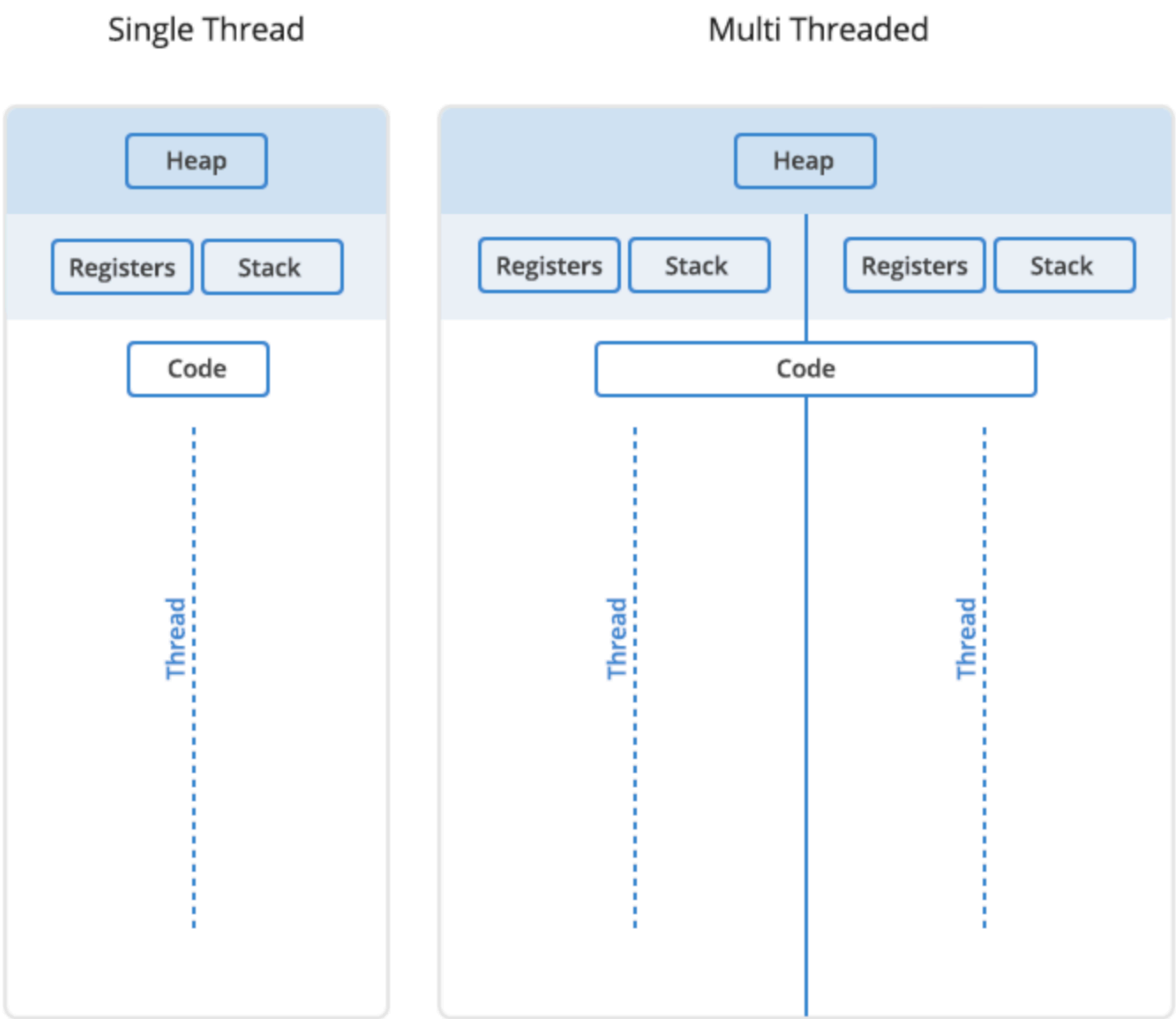
Concurrency

- Concurrent programs have multiple *threads of execution* (a.k.a. *threads*)
- In most runtime environments:
  - OS acts as scheduler to determine when thread executes its next instruction
  - Each process can have multiple threads
    - Which share resources, such as memory
    - Because scheduler decides when threads execute, programmer cannot rely on their ordering
      - So synchronization often required

Concurrency

A Computer Process




Single Thread
Multi Threaded
Heap
Registers
Stack
Code
Thread
Heap
Registers
Stack
Registers
Stack
Code
Thread
Thread

Concurrency

Processes vs. Threads — Advantages and Disadvantages

PROCESS	THREAD
Processes are heavyweight operations	Threads are lighter weight operations
Each process has its own memory space	Threads use the memory of the process they belong to
Inter-process communication is slow as processes have different memory addresses	Inter-thread communication can be faster than inter-process communication because threads of the same process share memory with the process they belong to
Context switching between processes is more expensive	Context switching between threads of the same process is less expensive
Processes don't share memory with other processes	Threads share memory with other threads of the same process

Concurrency

- The tradeoff: programs can execute multiple tasks in the same time period
  - Which can result in serious speedup if run on a multi-core processor or other concurrent hardware
- In general: programmer initializes threads, starts them running, then deals with results as they are returned
  - Sort of like sending off minions (threads) to do your work

Concurrency in Modern C++

- First, thorough treatment requires an entire book
  - We just give a short intro
- In modern C++, achieve concurrency by creating asynchronous tasks
  - A task that does not immediately need a result
- To launch, use `std::async` function template in the `<future>` header

Aside: Variadic Functions

- *Variadic functions* take a variable number of arguments
  - E.g., `printf` – you provide format specifier and variable number of parameters
  - Variadic functions declared by placing … as the final parameter
  - On invocation, compiler matches supplied parameters against declared arguments. Remainder are represented by …

Variadic Functions

- *Variadic functions* take a variable number of arguments

```
int sum(size_t n, ...) {
```

- Extract individual arguments from variadic arguments via utility functions in the `<cstdarg>` header

Variadic Functions

- *Variadic functions* take a variable number of arguments

This is actual C++ syntax, not slide shorthand.

```
int sum(size_t n, ...) {
```

- Extract individual arguments from variadic arguments via utility functions in the `<cstdarg>` header

Variadic Functions

Table 9-1: Utility Functions in the <cstdarg> Header

Function	Description
va_list	Used to declare a local variable representing the variadic arguments
va_start	Enables access to the variadic arguments
va_end	Used to end iteration over the variadic arguments
va_arg	Used to iterate over each element in the variadic arguments
va_copy	Makes a copy of the variadic arguments

Variadic Functions

```
#include <cstdarg>
#include <cstdint>
#include <cstdio>

int sum(size_t n, ...) {
  va_list args;
  va_start(args, n);
  int result{};
  while(n--) {
    auto next_element = va_arg(args, int);
    result += next_element;
  }
  va_end(args);
  return result;
}

int main() {
  printf("The answer is %d.", sum(6, 2, 4, 6, 8, 10, 12));
}
```

Variadic Functions

```
#include <cstdarg>
#include <cstdint>
#include <cstdio>

int sum(size_t n, ...) {
  va_list args;
  va_start(args, n);
  int result{};
  while(n--) {
    auto next_element = va_arg(args, int);
    result += next_element;
  }
  va_end(args);
  return result;
}

int main() {
  printf("The answer is %d.", sum(6, 2, 4, 6, 8, 10, 12));
}
```

All variadic functions must declare a `va_list`. Here it's called `args`

Variadic Functions

```
#include <cstdarg>
#include <cstdint>
#include <cstdio>

int sum(size_t n, ...) {
  va_list args;
  va_start(args, n);
  int result{};
  while(n--) {
    auto next_element = va_arg(args, int);
    result += next_element;
  }
  va_end(args);
  return result;
}

int main() {
  printf("The answer is %d.", sum(6, 2, 4, 6, 8, 10, 12));
}
```

A `va_list` requires initialization with `va_start`. First argument to `va_start` is a va_list. Second is the number of variadic args.

Variadic Functions

```
#include <cstdarg>
#include <cstdint>
#include <cstdio>

int sum(size_t n, ...) {
  va_list args;
  va_start(args, n);
  int result{};
  while(n--) {
    auto next_element = va_arg(args, int);
    result += next_element;
  }
  va_end(args);
  return result;
}

int main() {
  printf("The answer is %d.", sum(6, 2, 4, 6, 8, 10, 12));
}
```

Iterate over `va_list` using the `va_arg` function. First argument to `va_arg` is the va_list. Second is the argument type.

Variadic Functions

```
#include <cstdarg>
#include <cstdint>
#include <cstdio>

int sum(size_t n, ...) {
  va_list args;
  va_start(args, n);
  int result{};
  while(n--) {
    auto next_element = va_arg(args, int);
    result += next_element;
  }
  va_end(args);
  return result;
}

int main() {
  printf("The answer is %d.", sum(6, 2, 4, 6, 8, 10, 12));
}
```

Once completed iterating, call `va_end` with the `va_list` structure.

Variadic Functions

- Variadic functions are a holdover from C
- Generally considered unsafe and a security vulnerability
- Two major problems:
  - Not type safe (note second argument to `va_args` is a type)
  - Number of elements in variadic arguments must be tracked separately
  - Compiler is no help with either

Variadic Functions

- Variadic *templates* are safer and better performing method for implementing variadic functions
  - I’ll leave that for your own study

Concurrency in Modern C++

- First, thorough treatment requires an entire book
  - We just give a short intro
- In modern C++, achieve concurrency by creating asynchronous tasks
  - A task that does not immediately need a result
- To launch, use `std::async` function template in the `<future>` header

Concurrency in Modern C++

- Simplified `async` declaration

```
std::future<FuncReturnType> std::async([policy], func, Args&&... args);
```

- First argument, which is optional, is the launch policy, `std::launch`
  - `std::launch::async` runtime creates a new thread to launch your task
  - `std::launch::deferred` runtime waits until you need task result before executing
    - *lazy evaluation*

Concurrency in Modern C++

```
std::future<FuncReturnType> std::async([policy], func, Args&&... args);
```

- First argument, which is optional, is the launch policy, `std::launch`
  - `std::launch::async` runtime creates a new thread to launch your task
  - `std::launch::deferred` runtime waits until you need task result before executing
  - Optional launch policy defaults to async|deferred
    - Meaning it's implementation dependent

Concurrency in Modern C++

```
std::future<FuncReturnType> std::async([policy], func, Args&&... args);
```

- Second argument: a function object representing task you want to execute
  - No restriction on number or type of arguments the function object accepts
  - And it might return any type

Concurrency in Modern C++

```
std::future<FuncReturnType> std::async([policy], func, Args&&... args);
```

- `std::async` is a variadic template with a function *parameter pack*
  - Bottom line: any arguments you pass beyond function object are used to invoke the function object when the task is launched
- `std::async` returns a `std::future` object

Concurrency in Modern C++

```
std::future<FuncReturnType> std::async([policy], func, Args&&... args);
```

- A `future` is a template that holds the value of an asynchronous task
  - It has a single parameter: the type of the asynchronous task's return value
  - E.g., if you pass a function object that returns a `string`, `async` will return a `future<string>`

Concurrency in Modern C++

- Given a `future`, you can interact with an asynchronous task in three ways:
  - Query the `future` about its validity
  - Obtain the value from the `future` using the `get()` method
  - Check whether a task has completed

Query A future About Its Validity

- A valid `future` has a shared state associated with it
  - So they can communicate the results of the task
- Any `future` returned by `async` is valid until you retrieve the asynchronous task's return value
  - At which point shared state's lifetime ends

Query A future About Its Validity

```
#include "catch2/catch.hpp"
#include <future>
#include <string>

using namespace std;

TEST_CASE("async returns valid future") {
  using namespace literals::string_literals;

  auto the_future = async([] { return "female"s; });
  REQUIRE(the_future.valid());
}
```

Query A future About Its Validity

```
#include "catch2/catch.hpp"
#include <future>
#include <string>

using namespace std;

TEST_CASE("async returns valid future") {
  using namespace literals::string_literals;

  auto the_future = async([] { return "female"s; });
  REQUIRE(the_future.valid());
}
```

You may be asking: What's with this thing? It's actually a constructor for a `string`. It's an example of operator overloading

```
std::literals::string_literals::operator""s
```

Query A future About Its Validity

```
#include "catch2/catch.hpp"
#include <future>
#include <string>

using namespace std;

TEST_CASE("async returns valid future") {
  using namespace literals::string_literals;

  auto the_future = async([] { return "female"s; });
  REQUIRE(the_future.valid());
}
```

The big difference (aside from notational convenience) is that a string constructed with this operator can include null characters inside the string

```
std::literals::string_literals::operator""s
```

Example operator""s

```
#include <string>
#include <iostream>

int main()
{
    using namespace std::string_literals;

    std::string s1 = "abc\0\0def";
    std::string s2 = "abc\0\0def"s;
    std::cout << "s1: " << s1.size() << " \"" << s1 << "\"\n";
    std::cout << "s2: " << s2.size() << " \"" << s2 << "\"\n";
}
```

Possible output:

```
s1: 3 "abc"
s2: 8 "abc^@^@def"
```

Thanks cppreference.com

Query A future About Its Validity

- Launch an asynchronous task that simply returns a `string`

```
#include "catch2/catch.hpp"
#include <future>
#include <string>

using namespace std;

TEST_CASE("async returns valid future") {
  using namespace literals::string_literals;

  auto the_future = async([] { return "female"s; });
  REQUIRE(the_future.valid());
}
```

- Because `async` always returns a valid `future`, `valid()` returns `true`

Query A future About Its Validity

- If you default construct a `future`, `valid()` will return `false`

```
TEST_CASE("future invalid by default") {
  future<bool> default_future;
  REQUIRE_FALSE(default_future.valid());
}
```

Obtain the Value from a `future`

- Obtain the value from the `future` using the `get()` method
- If the asynchronous task has not yet completed, the call to `get()` will block the currently executed thread until the result is available

Obtain the Value from a `future`

- Obtain the value from the `future` using the `get()` method

```
TEST_CASE("get returns value") {
  using namespace literals::string_literals;

  auto the_future = async([] { return "female"s; });
  REQUIRE(the_future.get() == "female");
}
```

- Task is launched using call to `asycn`. Results is obtained from returned `future`

Obtain the Value from a `future`

- If an asynchronous task throws an exception, the `future` will collect it and throw it when `get()` is called

```
TEST_CASE("get may throw ") {
  auto ghostrider = async([] { throw runtime_error{ "The pattern is full." }; });
  REQUIRE_THROWS_AS(ghostrider.get(), runtime_error);
}
```

Aside: The `stdlib` `Chrono` Library

- Provides a variety of clocks in the <chrono> header
- Useful for when you want to program something that depends on time or for timing your code
- Provides three clocks, all in the `std::chrono` namespace, with each providing a different guarantee

Aside: The `stdlib Chrono` Library

- `std::chrono::system_clock` is the system wide real-time clock
  - A.K.A. the *wall clock*
  - Provides elapsed time since an implementation specific start date
    - Most use January 1, 1970 at midnight

Aside: The `stdlib` `Chrono` Library

- `std::chrono::steady_clock` guarantees that its value will never decrease
  - Might seem absurd, but measuring time is complicated -- might have to deal with leap seconds and/or inaccurate clocks
- Aside: I once had to deal with real-world situation where triangle inequality failed!
  - So yes, this kind of stuff happens

Aside: The `stdlib` `Chrono` Library

- `std::chrono::high_resolution_clock` has the shortest *tick* period available
  - tick is the smallest atomic change that the clock can measure
    - I.e., the granularity of the clock
- Beware of situations where tick is, say, millisecond, but clock is only updated every half second!
  - Mostly a historical issue now

Aside: The `stdlib` `Chrono` Library

- Each clock supports the static member function `now()`, which returns a *time point* corresponding to the current value of the clock
- time point represents a moment in time
- `chrono` encodes time points using `std::chrono::time_point` type

Aside: The `stdlib` `Chrono` Library

- Using `time_point` objects is relatively easy
- They provide a `time_since_epoch()` method that returns the amount of time lapsed between the `time_point` and the clock's *epoch*
- This elapsed time is called a *duration*

Aside: The `stdlib Chrono` Library

- epoch is an implementation defined reference point denoting the beginning of the clock
- UNIX epoch (or POSIX time) begins on January 1, 1970
- Windows epoch begins January 1, 1601
  - Corresponding to beginning of a 400 year Gregorian-calendar cycle

Aside: The `stdlib` `Chrono` Library

- An alternate method to obtain a duration from a `time_point` is to subtract two of them
- A `std::chrono:duration` represents the time between two `time_point` objects
- Durations expose a `count()` method that returns the number of clock ticks in the duration

Aside: The `stdlib Chrono` Library

```
TEST_CASE("chrono supports several clocks") {
  auto sys_now = std::chrono::system_clock::now();
  REQUIRE(sys_now.time_since_epoch().count() > 0);

  auto hires_now = std::chrono::high_resolution_clock::now();
  REQUIRE(hires_now.time_since_epoch().count() > 0);

  auto steady_now = std::chrono::steady_clock::now();
  REQUIRE(steady_now.time_since_epoch().count() > 0);
}
```

- Each of the `auto` variables are `time_point` objects. And each of these exposes the `time_since_epoch()` method

Aside: The `stdlib` `Chrono` Library

```
TEST_CASE("chrono supports several clocks") {
  auto sys_now = std::chrono::system_clock::now();
  REQUIRE(sys_now.time_since_epoch().count() > 0);

  auto hires_now = std::chrono::high_resolution_clock::now();
  REQUIRE(hires_now.time_since_epoch().count() > 0);

  auto steady_now = std::chrono::steady_clock::now();
  REQUIRE(steady_now.time_since_epoch().count() > 0);
}
```

- `time_since_epoch()` returns a `duration`, and the `count()` method of that `duration` returns the number of ticks

Aside: The `stdlib` `Chrono` Library

Any clock has a `now()` method

`now()` → `time_point`

any `time_point` has a `time_since_epoch()` method

`time_since_epoch()` → `duration`

Any `duration` has a `count()` method → number of ticks

Aside: The `stdlib` `Chrono` Library

- `duration` objects can also be constructed directly
- `std::chrono` namespace contains helper functions for generating durations
- `std::chrono::chrono_literals` namespace offers User-defined literals for creating durations

Aside: The `stdlib Chrono` Library

Helper function	Literal equivalent
`nanoseconds(3600000000000)`	`3600000000000ns`
`microseconds(3600000000)`	`3600000000us`
`milliseconds(3600000)`	`3600000ms`
`seconds(3600)`	`3600s`
`minutes(60)`	`60m`
`hours(1)`	`1h`

Note you don't have to use those exact numerical values. Also, for example, `ms` is similar to appending `L` to a long value

Aside: The `stdlib` `Chrono` Library

```cpp
#include <chrono>
TEST_CASE("chrono supports several units of measurement") {
  using namespace std::literals::chrono_literals;
  auto one_s = std::chrono::seconds(1);
  auto thousand_ms = 1000ms;
  REQUIRE(one_s == thousand_ms);
}
```

Aside: The `stdlib` `Chrono` Library

- Chrono also supplies the function template `std::chrono::duration_cast` which does pretty much what you'd expect: converts a duration from one unit to another (e.g., seconds to minutes)
  - And it works, pretty much how you'd expect

Aside: The stdlib Chrono Library

- `std::chrono::duration_cast`

```
TEST_CASE("chrono supports duration_cast") {
  using namespace std::chrono;
  auto billion_ns_as_s = duration_cast<seconds>(1000000000ns);
  REQUIRE(billion_ns_as_s.count() == 1);
}
```

What you want to cast

What you want to cast to

Aside: The `stdlib Chrono` Library

- Waiting: You can use durations to specify an amount of time for your program to wait
- stdlib provides additional concurrency primitives in the `<threads>` header
  - Contains the non-member function `std::this_thread::sleep_for`
  - `sleep_for` accepts a `duration` argument corresponding to how long you want your thread to wait (or "sleep")

Aside: The stdlib Chrono Library

```
#include <thread>
#include <chrono>
TEST_CASE("chrono used to sleep") {
  using namespace std::literals::chrono_literals;
  auto start = std::chrono::system_clock::now();
  std::this_thread::sleep_for(100ms);
  auto end = std::chrono::system_clock::now();
  REQUIRE(end - start >= 100ms);
}
```

So Let's Use This

- Optimizing code requires accurate measurement (to determine how long a particular code path takes)
- Chrono is very useful for this
- The `Stopwatch` class defined in the following (user defined, not in a standard library) is an example of how you can measure time in a code path
- The idea: a `Stopwatch` object keeps a reference to a `duration` object

So Let's Use This

- When the `Stopwatch` is constructed, the time (via `now()`) is recorded
- When the `Stopwatch` is destructed, the time since the start is recorded
- So, construct your `Stopwatch`, run your task, destruct your `Stopwatch`

Stopwatch

```
struct Stopwatch {
  Stopwatch(std::chrono::nanoseconds& result)
      : result{ result }
      , start{ std::chrono::high_resolution_clock::now() } {}
  ~Stopwatch() {
    result = std::chrono::high_resolution_clock::now() - start;
  }

  private:
  std::chrono::nanoseconds& result;
  const std::chrono::time_point<std::chrono::high_resolution_clock> start;
};
```

- The `result` instance variable is a reference to a `duration` (with nanosecond granularity)
- start is a `time_point` for a `high_resolution_clock`

Stopwatch

```
struct Stopwatch {
  Stopwatch(std::chrono::nanoseconds& result)
      : result{ result }
      , start{ std::chrono::high_resolution_clock::now() } {}
  ~Stopwatch() {
    result = std::chrono::high_resolution_clock::now() - start;
  }

  private:
  std::chrono::nanoseconds& result;
  const std::chrono::time_point<std::chrono::high_resolution_clock> start;
};
```

- When the `Stopwatch` is constructed, `result` parameter is assigned to the `result` instance variable
- the time (via `now()`) is recorded

Stopwatch

```
struct Stopwatch {
  Stopwatch(std::chrono::nanoseconds& result)
      : result{ result }
      , start{ std::chrono::high_resolution_clock::now() } {}
  ~Stopwatch() {
    result = std::chrono::high_resolution_clock::now() - start;
  }

  private:
  std::chrono::nanoseconds& result;
  const std::chrono::time_point<std::chrono::high_resolution_clock> start;
};
```

- When the `Stopwatch` is destructed, `result` is assigned a `duration` that records the different between the current time and `start`
  - Current time is obtained via `now()`

Using Stopwatch

```
#include <chrono>
#include <cstdio>

struct Stopwatch {
  Stopwatch(std::chrono::nanoseconds& result)
      : result{ result }
      , start{ std::chrono::system_clock::now() } {}
  ~Stopwatch() {
    result = std::chrono::system_clock::now() - start;
  }

  private:
  std::chrono::nanoseconds& result;
  const std::chrono::time_point<std::chrono::system_clock> start;
};

int main() {
  const size_t n = 100'000'000;
  std::chrono::nanoseconds elapsed;
  {
    Stopwatch stopwatch{ elapsed };
    volatile double result{ 1.23e45 };
    for (double i = 1; i < n; i++) {
      result /= i;
    }
  }
  auto time_per_addition = elapsed.count() / double{ n };
  printf("Took %gns per division.", time_per_addition);
}
```

What's with the apostrophes?

Using Stopwatch

```
#include <chrono>
#include <cstdio>

struct Stopwatch {
  Stopwatch(std::chrono::nanoseconds& result)
      : result{ result }
      , start{ std::chrono::system_clock::now() } {}
  ~Stopwatch() {
    result = std::chrono::system_clock::now() - start;
  }

  private:
  std::chrono::nanoseconds& result;
  const std::chrono::time_point<std::chrono::system_clock> start;
};

int main() {
  const size_t n = 100'000'000;
  std::chrono::nanoseconds elapsed;
  {
    Stopwatch stopwatch{ elapsed };
    volatile double result{ 1.23e45 };
    for (double i = 1; i < n; i++) {
      result /= i;
    }
  }
  auto time_per_addition = elapsed.count() / double{ n };
  printf("Took %gns per division.", time_per_addition);
}
```

What's with the parentheses? (Hint: it's not a method body)

Using `Stopwatch`

```
#include <chrono>
#include <cstdio>

struct Stopwatch {
  Stopwatch(std::chrono::nanoseconds& result)
      : result{ result }
      , start{ std::chrono::system_clock::now() } {}
  ~Stopwatch() {
    result = std::chrono::system_clock::now() - start;
  }

  private:
  std::chrono::nanoseconds& result;
  const std::chrono::time_point<std::chrono::system_clock> start;
};

int main() {
  const size_t n = 100'000'000;
  std::chrono::nanoseconds elapsed;
  {
    Stopwatch stopwatch{ elapsed };
    volatile double result{ 1.23e45 };
    for (double i = 1; i < n; i++) {
      result /= i;
    }
  }
  auto time_per_addition = elapsed.count() / double{ n };
  printf("Took %gns per division.", time_per_addition);
}
```

What's with the `volatile` keyword?

volatile

```
int main() {
  const size_t n = 100'000'000;
  std::chrono::nanoseconds elapsed;
  {
    Stopwatch stopwatch{ elapsed };
    volatile double result{ 1.23e45 };
    for (double i = 1; i < n; i++) {
      result /= i;
    }
  }
  auto time_per_addition = elapsed.count() / double{ n };
  printf("Took %gns per division.\n", time_per_addition);
}
```

- According to the standard: [..] `volatile` is a hint to the implementation to **avoid aggressive optimization involving the object** because the value of the object might be changed by means undetectable by an implementation.[...]

volatile

```
int main() {
  const size_t n = 100'000'000;
  std::chrono::nanoseconds elapsed;
  {
    Stopwatch stopwatch{ elapsed };
    volatile double result{ 1.23e45 };
    for (double i = 1; i < n; i++) {
      result /= i;
    }
  }
  auto time_per_addition = elapsed.count() / double{ n };
  printf("Took %gns per division.\n", time_per_addition);
}
```

- In English: The compiler can see that the value of `n` never changes, so it might try to optimize away the `for` loop (thus avoiding the conditional check on each iteration, which can involve fetching the value of the variable `i`, comparing to `n`, etc).

volatile

```
int main() {
  const size_t n = 100'000'000;
  std::chrono::nanoseconds elapsed;
  {
    Stopwatch stopwatch{ elapsed };
    volatile double result{ 1.23e45 };
    for (double i = 1; i < n; i++) {
      result /= i;
    }
  }
  auto time_per_addition = elapsed.count() / double{ n };
  printf("Took %gns per division.\n", time_per_addition);
}
```

- In English: volatile says ”Don’t do this. Though it looks like the value of `n` never changes, it may actually at times change through means of which you may not be aware and/or cannot detect.”

volatile

```
int main() {
  const size_t n = 100'000'000;
  std::chrono::nanoseconds elapsed;
  {
    Stopwatch stopwatch{ elapsed };
    volatile double result{ 1.23e45 };
    for (double i = 1; i < n; i++) {
      result /= i;
    }
  }
  auto time_per_addition = elapsed.count() / double{ n };
  printf("Took %gns per division.\n", time_per_addition);
}
```

- In this particular example, we're trying to time the iterations of the loop, so we don't want the loop to be optimized out of the executable code. Since `result` is declared `volatile`, and appears in the loop, the compiler will not optimize out the loop.

Thanks to StackOverflow: https://stackoverflow.com/questions/4437527/why-do-we-use-volatile-keyword

Back to the `futures`

Check Whether an Asynchronous Task Has Completed

- Use `std::wait_for` if you have a `duration` object
- Use `std::wait_until` if you have a `time_point` object
- Both return a `std::future_status`

Check Whether an Asynchronous Task Has Completed

- `std::future_status` can have one of three values
  - `future_status::deferred` task will be evaluated lazily, so task will execute once you call `get()`
  - `future_status::ready` task has completed and result is ready
  - `future_status::timeout` task is not ready
- If task completes before assigned waiting period, `async` will return early

An Example Using `wait_for`

```
TEST_CASE("wait_until indicates whether a task is ready") {
  using namespace literals::chrono_literals;
  auto sleepy = async(launch::async, [] { this_thread::sleep_for(100ms); });
  const auto not_ready_yet = sleepy.wait_for(25ms);
  REQUIRE(not_ready_yet == future_status::timeout);
  const auto totally_ready = sleepy.wait_for(100ms);
  REQUIRE(totally_ready == future_status::ready);
}
```

An Example Using `wait_for`

```
TEST_CASE("wait_until indicates whether a task is ready") {
  using namespace literals::chrono_literals;
  auto sleepy = async(launch::async, [] { this_thread::sleep_for(100ms); });
  const auto not_ready_yet = sleepy.wait_for(25ms);
  REQUIRE(not_ready_yet == future_status::timeout);
  const auto totally_ready = sleepy.wait_for(100ms);
  REQUIRE(totally_ready == future_status::ready);
}
```

- First, a task launched with `asycn`, which just waits for 100ms before returning
- Next, call `wait_for` with 25ms. Because 25ms is less than 100ms, we expect that task is still sleeping, so `wait_for` returns `future_status::timeout`.
- Call `wait_for` again and wait for up to another 100ms.
- Because second wait_for will finish after task, `wait_for` returns a `future_status::ready`

An Example Using `wait_for`

```
TEST_CASE("wait_until indicates whether a task is ready") {
  using namespace literals::chrono_literals;
  auto sleepy = async(launch::async, [] { this_thread::sleep_for(100ms); });
  const auto not_ready_yet = sleepy.wait_for(25ms);
  REQUIRE(not_ready_yet == future_status::timeout);
  const auto totally_ready = sleepy.wait_for(100ms);
  REQUIRE(totally_ready == future_status::ready);
}
```

- Technically, these assertions are not guaranteed to pass. `this_thread::sleep_for` is not exact. The OS is responsible for scheduling threads. It might schedule the sleeping thread later than the specified duration.

An Example: Factoring

First: Doing it serially
Second: Doing it with threads

```cpp
#include <array>
#include <chrono>
#include <iostream>
#include <limits>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

struct Stopwatch {
  Stopwatch(std::chrono::nanoseconds& result)
      : result{ result }
      , start{ std::chrono::high_resolution_clock::now() } {}
  ~Stopwatch() {
    result = std::chrono::high_resolution_clock::now() - start;
  }

  private:
  std::chrono::nanoseconds& result;
  const std::chrono::time_point<std::chrono::high_resolution_clock> start;
};

template <typename T>
vector<T> factorize(T x) {
  vector<T> result{ 1 };
  for(T candidate = 2; candidate <= x; candidate++) {
    if(x % candidate == 0) {
      result.push_back(candidate);
      x /= candidate;
      candidate = 1;
    }
  }
  return result;
}
```

```cpp
#include <array>
#include <chrono>
#include <iostream>
#include <limits>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

struct Stopwatch {
  Stopwatch(std::chrono::nanoseconds& result)
      : result{ result }
      , start{ std::chrono::high_resolution_clock::now() } {}
  ~Stopwatch() {
    result = std::chrono::high_resolution_clock::now() - start;
  }

  private:
  std::chrono::nanoseconds& result;
  const std::chrono::time_point<std::chrono::high_resolution_clock> start;
};

template <typename T>
vector<T> factorize(T x) {
  vector<T> result{ 1 };
  for(T candidate = 2; candidate <= x; candidate++) {
    if(x % candidate == 0) {
      result.push_back(candidate);
      x /= candidate;
      candidate = 1;
    }
  }
  return result;
}
```

Note that this is NOT an efficient factoring algorithm!

```
string factor_task(unsigned long long x) {
  chrono::nanoseconds elapsed_ns;
  vector<unsigned long long> factors;
  {
    Stopwatch stopwatch{ elapsed_ns };
    factors = factorize(x);
  }
  const auto elapsed_ms = chrono::duration_cast<chrono::milliseconds>(elapsed_ns).count();
  stringstream ss;
  ss << elapsed_ms << " ms: Factoring " << x << " ( ";
  for(auto factor : factors)
    ss << factor << " ";
  ss << ")\n";
  return ss.str();
}


array<unsigned long long, 6> numbers{ 9699690,     179426549,   1000000007,
                                      4294967291, 4294967296, 1307674368000 };

int main() {
  chrono::nanoseconds elapsed_ns;
  {
    Stopwatch stopwatch{ elapsed_ns };
    for(auto number : numbers)
      cout << factor_task(number);
  }
  const auto elapsed_ms = chrono::duration_cast<chrono::milliseconds>(elapsed_ns).count();
  cout << elapsed_ms << "ms: total program time\n";
}
```

```cpp
string factor_task(unsigned long long x) {
  chrono::nanoseconds elapsed_ns;
  vector<unsigned long long> factors;
  {
    Stopwatch stopwatch{ elapsed_ns };
    factors = factorize(x);
  }
  const auto elapsed_ms = chrono::duration_cast<chrono::milliseconds>(elapsed_ns).count();
  stringstream ss;
  ss << elapsed_ms << " ms: Factoring " << x << " ( ";
  for(auto factor : factors)
    ss << factor << " ";
  ss << ")\n";
  return ss.str();
}


array<unsigned long long, 6> numbers{ 9699690,     179426549,   1000000007,
                                      4294967291, 4294967296, 1307674368000 };

int main() {
  chrono::nanoseconds elapsed_ns;
  {
    Stopwatch stopwatch{ elapsed_ns };
    for(auto number : numbers)
      cout << factor_task(number);
  }
  const auto elapsed_ms = chrono::duration_cast<chrono::milliseconds>(elapsed_ns).count();
  cout << elapsed_ms << "ms: total program time\n";
}
```

```
0 ms: Factoring 9699690 ( 1 2 3 5 7 11 13 17 19 )
1284 ms: Factoring 179426549 ( 1 179426549 )
7156 ms: Factoring 1000000007 ( 1 1000000007 )
30439 ms: Factoring 4294967291 ( 1 4294967291 )
0 ms: Factoring 4294967296 ( 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 )
0 ms: Factoring 1307674368000 ( 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 5 5 5 7 7 11 13 )
38880ms: total program time
```

```cpp
#include <array>
#include <chrono>
#include <future>
#include <iostream>
#include <limits>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

struct Stopwatch {
  Stopwatch(std::chrono::nanoseconds& result)
      : result{ result }
      , start{ std::chrono::high_resolution_clock::now() } {}
  ~Stopwatch() {
    result = std::chrono::high_resolution_clock::now() - start;
  }

  private:
  std::chrono::nanoseconds& result;
  const std::chrono::time_point<std::chrono::high_resolution_clock> start;
};

template <typename T>
vector<T> factorize(T x) {
  vector<T> result{ 1 };
  for(T candidate = 2; candidate <= x; candidate++) {
    if(x % candidate == 0) {
      result.push_back(candidate);
      x /= candidate;
      candidate = 1;
    }
  }
  return result;
}
```

```
string factor_task(unsigned long long x) {
  chrono::nanoseconds elapsed_ns;
  vector<unsigned long long> factors;
  {
    Stopwatch stopwatch{ elapsed_ns };
    factors = factorize(x);
  }
  const auto elapsed_ms = chrono::duration_cast<chrono::milliseconds>(elapsed_ns).count();
  stringstream ss;
  ss << elapsed_ms << " ms: Factoring " << x << " ( ";
  for(auto factor : factors)
    ss << factor << " ";
  ss << ")\n";
  return ss.str();
}

array<unsigned long long, 6> numbers{ 9699690,     179426549,    1000000007,
                                      4294967291, 4294967296, 1307674368000 };

int main() {
  chrono::nanoseconds elapsed_ns;
  {
    Stopwatch stopwatch{ elapsed_ns };
    vector<future<string>> factor_tasks;
    for(auto number : numbers)
      factor_tasks.emplace_back(async(launch::async, factor_task, number));
    for(auto& task : factor_tasks)
      cout << task.get();
  }
  const auto elapsed_ms = chrono::duration_cast<chrono::milliseconds>(elapsed_ns).count();
  cout << elapsed_ms << "ms : total program time\n";
}
```

```cpp
string factor_task(unsigned long long x) {
  chrono::nanoseconds elapsed_ns;
  vector<unsigned long long> factors;
  {
    Stopwatch stopwatch{ elapsed_ns };
    factors = factorize(x);
  }
  const auto elapsed_ms = chrono::duration_cast<chrono::milliseconds>(elapsed_ns).count();
  stringstream ss;
  ss << elapsed_ms << " ms: Factoring " << x << " ( ";
  for(auto factor : factors)
    ss << factor << " ";
  ss << ")\n";
  return ss.str();
}

array<unsigned long long, 6> numbers{ 9699690,     179426549,   1000000007,
                                      4294967291, 4294967296, 1307674368000 };

int main() {
  chrono::nanoseconds elapsed_ns;
  {
    Stopwatch stopwatch{ elapsed_ns };
    vector<future<string>> factor_tasks;
    for(auto number : numbers)
      factor_tasks.emplace_back(async(launch::async, factor_task, number));
    for(auto& task : factor_tasks)
      cout << task.get();
  }
  const auto elapsed_ms = chrono::duration_cast<chrono::milliseconds>(elapsed_ns).count();
  cout << elapsed_ms << "ms : total program time\n";
}
```

```
0 ms: Factoring 9699690 ( 1 2 3 5 7 11 13 17 19 )
1256 ms: Factoring 179426549 ( 1 179426549 )
6950 ms: Factoring 1000000007 ( 1 1000000007 )
29608 ms: Factoring 4294967291 ( 1 4294967291 )
0 ms: Factoring 4294967296 ( 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 )
0 ms: Factoring 1307674368000 ( 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 5 5 7 7 11 13 )
29608ms : total program time
```

So, concurrent programming is easy, right?

So, concurrent programming is easy, right?

Only if your threads don't have to be synchronized and don't involve sharing mutable data…

```cpp
#include <future>
#include <iostream>

using namespace std;

void goat_rodeo() {
  const size_t iterations{ 1'000'000 };
  int tin_cans_available{};
  auto eat_cans = async(launch::async, [&] {
    for(size_t i{}; i < iterations; i++)
      tin_cans_available--;
  });
  auto deposit_cans = async(launch::async, [&] {
    for(size_t i{}; i < iterations; i++)
      tin_cans_available++;
  });
  eat_cans.get();
  deposit_cans.get();
  cout << "Tin cans: " << tin_cans_available << "\n";
}

int main() {
  goat_rodeo();
  goat_rodeo();
  goat_rodeo();
}
```

```cpp
#include <future>
#include <iostream>

using namespace std;

void goat_rodeo() {
  const size_t iterations{ 1'000'000 };
  int tin_cans_available{};
  auto eat_cans = async(launch::async, [&] {
    for(size_t i{}; i < iterations; i++)
      tin_cans_available--;
  });
  auto deposit_cans = async(launch::async, [&] {
    for(size_t i{}; i < iterations; i++)
      tin_cans_available++;
  });
  eat_cans.get();
  deposit_cans.get();
  cout << "Tin cans: " << tin_cans_available << "\n";
}

int main() {
  goat_rodeo();
  goat_rodeo();
  goat_rodeo();
}
```

What the heck are these?!

Do you ever get the feeling that every time I show you a code example I also have to explain another aspect of C++?

Do you ever get the feeling that every time I show you a code example I also have to explain another aspect of C++?

If so, you're right. There is a lot to this language! So…

Recall: Lambda Captures

lambda version of `CountIf`

```
#include <cstdint>
#include <cstdio>

int main() {
  char to_count{ 's' };
  auto s_counter = [to_count](const char* str) {
    size_t index{}, result{};
    while(str[index]) {
      if(str[index] == to_count)
        result++;
      index++;
    }
    return result;
  };
  auto sally = s_counter("Sally sells seashells by the seashore.");
  printf("Sally: %zd\n", sally);
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zd\n", sailor);
}
```

`to_count` captured and can now be used within lambda's body

Lambda Captures

- Lambda captures can be used to make available to the lambda any local variables in the procedure in which the lamda appears (they can be used within the lambda body)
- To capture all of the local variables by value, the syntax is [=]
- To capture all of the local variables by reference, the syntax is [&]

```
#include <future>
#include <iostream>

using namespace std;

void goat_rodeo() {
  const size_t iterations{ 1'000'000 };
  int tin_cans_available{};
  auto eat_cans = async(launch::async, [&] {
    for(size_t i{}; i < iterations; i++)
      tin_cans_available--;
  });
  auto deposit_cans = async(launch::async, [&] {
    for(size_t i{}; i < iterations; i++)
      tin_cans_available++;
  });
  eat_cans.get();
  deposit_cans.get();
  cout << "Tin cans: " << tin_cans_available << "\n";
}

int main() {
  goat_rodeo();
  goat_rodeo();
  goat_rodeo();
}
```

So now you know what these are: makes both local variables captured by value

You Might Think…

- That since `eat_cans()` (which decrements `tin_cans_available`) and `deposit_cans()` (which increments it) are both called the same number of times, that at the end, `tin_cans_available` would be zero…

You Might Think…

- That since `eat_cans()` (which decrements `tin_cans_available`) and `deposit_cans()` (which increments it) are both called the same number of times, that at the end, `tin_cans_available` would be zero…
- But you'd be wrong. The value of `tin_cans_available` at the end of the program is dependent on the exact order in which the instances of the two threads execute

You Might Think…

- But you'd be wrong. The value of `tin_cans_available` at the end of the program is dependent on the exact order in which the instances of the two threads execute
- And this varies from execution to execution in unpredictable ways
- This is called a *race condition*, because the result depends on which threads execute first

Let’s Run the Code

```
(base) m1-mcs-dszajda:chapter_19 dszajda$ ./goat_rodeo
Tin cans: -939312
Tin cans: -181226
Tin cans: 628864
```

So What Caused This?

- Note that in order to increment or decrement `tin_cans_available`, the variable first has to be read
  - Otherwise you can't know what you are incrementing or decrementing
- So sequence is "read, compute, write"
- In following use `cans_available` for space reasons

So What Caused This?

deposit_cans	eat_cans	cans_available
Read cans_available (0)		0
	Read cans_available (0)	0
Compute cans_available+1 (1)		0
	Compute cans_available–1 (–1)	0
Write cans_available+1 (1)		1
	Write cans_available–1 (–1)	–1

- Value in prens is result of task
- Note value of `cans_available` does not change until written

So What Caused This?

deposit_cans	eat_cans	cans_available
Read cans_available (0)		0
	Read cans_available (0)	0
Compute cans_available+1 (1)		0
	Compute cans_available–1 (–1)	0
Write cans_available+1 (1)		1
	Write cans_available–1 (–1)	–1

- The fundmental problem: Unsynchronized access to mutually shared data
  - Remember, at machine language level, instructions for reading, computing, writing, are separate

So What Can We Do?

- Synchronization primitives
- Three covered (briefly) in your text
  - mutexes
  - condition variables
  - locks
- Don’t think we’ll get to all of them, but we’ll see
  - Again, the goal in CS 240 is an introduction…

mutex

- The term *mutex* is short for *mutual exclusion algorithm*
- Mutexes support two operations:
  - Lock: When a thread needs to access shared data, it locks the mutex
    - Which can block the thread if another thread already has the lock
  - Unlock: When a thread no longer needs access to the data
- `<mutex>` header exposes several mutex options

mutex

- The term *mutex* is short for *mutual exclusion algorithm*
- `<mutex>` header exposes several mutex options
  - Ex: std::mutex -- basic mutual exclusion
  - Ex. std::timed_mutex – mutual exclusion with a timeout
    - If the mutex is not available by the specified `duration` or `time_point`, return
  - Lot’s more. We’ll only cover `std::mutex`

mutex

- `mutex` has only a single default constructor
- To obtain mutual exclusion, call either
  - `lock`: accepts no arguments and returns `void`. Thread blocks until `mutex` becomes available
  - `try_lock`: accepts no arguments and returns a `bool`. It returns immediately. If the `try_lock` successfully obtained mutual exclusion, it returns `true` and the calling thread now owns the lock. If not successful, it returns `false` and calling thread does not own the lock
- To release lock: call `unlock` (no args, returns `void`)

```
#include <future>
#include <iostream>
#include <mutex>

using namespace std;

void goat_rodeo() {
  const size_t iterations{ 1'000'000 };
  int tin_cans_available{};
  mutex tin_can_mutex;
  auto eat_cans = async(launch::async, [&] {
    for(size_t i; i < iterations; i++) {
      tin_can_mutex.lock();
      tin_cans_available--;
      tin_can_mutex.unlock();
    }
  });
  auto deposit_cans = async(launch::async, [&] {
    for(size_t i; i < iterations; i++) {
      tin_can_mutex.lock();
      tin_cans_available++;
      tin_can_mutex.unlock();
    }
  });
  eat_cans.get();
  deposit_cans.get();
  cout << "Tin cans: " << tin_cans_available << "\n";
}

int main() {
  goat_rodeo();
  goat_rodeo();
  goat_rodeo();
}
```

```cpp
#include <future>
#include <iostream>
#include <mutex>

using namespace std;

void goat_rodeo() {
  const size_t iterations{ 1'000'000 };
  int tin_cans_available{};
  mutex tin_can_mutex;
  auto eat_cans = async(launch::async, [&] {
    for(size_t i; i < iterations; i++) {
      tin_can_mutex.lock();
      tin_cans_available--;
      tin_can_mutex.unlock();
    }
  });
  auto deposit_cans = async(launch::async, [&] {
    for(size_t i; i < iterations; i++) {
      tin_can_mutex.lock();
      tin_cans_available++;
      tin_can_mutex.unlock();
    }
  });
  eat_cans.get();
  deposit_cans.get();
  cout << "Tin cans: " << tin_cans_available << "\n";
}

int main() {
  goat_rodeo();
  goat_rodeo();
  goat_rodeo();
}
```

Note that each thread acquires a lock before modifying `tin_cans_available`

```cpp
#include <future>
#include <iostream>
#include <mutex>

using namespace std;

void goat_rodeo() {
  const size_t iterations{ 1'000'000 };
  int tin_cans_available{};
  mutex tin_can_mutex;
  auto eat_cans = async(launch::async, [&] {
    for(size_t i; i < iterations; i++) {
      tin_can_mutex.lock();
      tin_cans_available--;
      tin_can_mutex.unlock();
    }
  });
  auto deposit_cans = async(launch::async, [&] {
    for(size_t i; i < iterations; i++) {
      tin_can_mutex.lock();
      tin_cans_available++;
      tin_can_mutex.unlock();
    }
  });
  eat_cans.get();
  deposit_cans.get();
  cout << "Tin cans: " << tin_cans_available << "\n";
}

int main() {
  goat_rodeo();
  goat_rodeo();
  goat_rodeo();
}
```

```
(base) m1-mcs-dszajda:chapter_19 dszajda$ ./goat_rodeo_locks
Tin cans: 0
Tin cans: 0
Tin cans: 0
```

How Are mutexes Implemented?

- Several ways
- One simple way: spin lock
  - Thread executes a loop until the lock is released
  - Advantage: usually minimizes amount of time between one thread releasing the lock and another acquiring it
  - Disadvantage (big): CPU is spending time checking for lock availability when another thread could be progressing

How Are mutexes Implemented?

- More modern (e.g., Windows)
- Mutexes based on *asynchronous procedure calls*
  - Roughly: thread waiting on mutex goes into a *wait state*. When lock becomes available, OS wakes up the waiting thread and hands off ownership of the lock
  - Advantage: other threads can progress while thread is waiting on lock

How Are mutexes Implemented?

- Usually: don’t need to worry about how mutexes are implemented on your system…
  - Unless the become a bottleneck in your program

A Problem…

- Suppose a thread acquires a lock, then fails to unlock
  - E.g., because the thread throws an exception
  - Then your program can halt
- Better alternative than manual handling of mutexes

Recall RAII

- You DO recall what RAII means?

Recall RAII

- Resource Acquisition Is Initialization
- General idea (and an important modern C++ programming principle): Bind the the life cycle of a resource that must be acquired (e.g. dynamic memory, mutex) to the lifetime of an object
- You do this when you acquire dynamic memory in a constructor and return it in a destructor

Recall RAII

- Resource Acquisition Is Initialization
- The Standard Library provides, in the `<mutex>` header, RAII class templates for handling mutexes
- Ex. `std::lock_guard`: a non-copyable, non-movable RAII wrapper that accepts a `mutex` in its constructor, where it calls `lock`. It then calls `unlock` in the destructor

`lock_guard`

- Basically, construct a `lock_guard` at the beginning of any scope where you need synchronization
- Safer than manual handling of synchronization
- And does not add any runtime cost over manual handling of mutexes
  - Though mutexes usually involve significant runtime costs, no matter how you handle them.

```cpp
#include <future>
#include <iostream>
#include <mutex>

using namespace std;

void goat_rodeo() {
  const size_t iterations{ 1'000'000 };
  int tin_cans_available{};
  mutex tin_can_mutex;
  auto eat_cans = async(launch::async, [&] {
    for(size_t i{}; i < iterations; i++) {
      lock_guard<mutex> guard{ tin_can_mutex };
      tin_cans_available--;
    }
  });
  auto deposit_cans = async(launch::async, [&] {
    for(size_t i{}; i < iterations; i++) {
      lock_guard<mutex> guard{ tin_can_mutex };
      tin_cans_available++;
    }
  });
  eat_cans.get();
  deposit_cans.get();
  cout << "Tin cans: " << tin_cans_available << "\n";
}

int main() {
  goat_rodeo();
  goat_rodeo();
  goat_rodeo();
}
```

```cpp
#include <future>
#include <iostream>
#include <mutex>

using namespace std;

void goat_rodeo() {
  const size_t iterations{ 1'000'000 };
  int tin_cans_available{};
  mutex tin_can_mutex;
  auto eat_cans = async(launch::async, [&] {
    for(size_t i{}; i < iterations; i++) {
      lock_guard<mutex> guard{ tin_can_mutex };
      tin_cans_available--;
    }
  });
  auto deposit_cans = async(launch::async, [&] {
    for(size_t i{}; i < iterations; i++) {
      lock_guard<mutex> guard{ tin_can_mutex };
      tin_cans_available++;
    }
  });
  eat_cans.get();
  deposit_cans.get();
  cout << "Tin cans: " << tin_cans_available << "\n";
}

int main() {
  goat_rodeo();
  goat_rodeo();
  goat_rodeo();
}
```

Note `lock_guard` is a parametrized type

Aside: `time`

- Yes, the Stopwatch we built is nice for seeing how long a code path takes to execute
- But sometimes you just want to know how long an entire program takes
- An in Linux, there is a nice command for doing that: `time`

Aside: `time`

- Just type time followed by the program/command on the command line and time will provide you with three values:
  - real: total time taken by program/command
  - user: time taken by program in user mode
  - sys: time taken by program in kernel mode

```
(base) m1-mcs-dszajda:chapter_19 dszajda$ time ./goat_rodeo
Tin cans: -780816
Tin cans: -718626
Tin cans: -872537

real	0m0.026s
user	0m0.028s
sys	0m0.003s
(base) m1-mcs-dszajda:chapter_19 dszajda$ time ./goat_rodeo_locks
Tin cans: 0
Tin cans: 0
Tin cans: 0

real	0m0.293s
user	0m0.197s
sys	0m0.264s
(base) m1-mcs-dszajda:chapter_19 dszajda$ time ./goat_rodeo_guards
Tin cans: 0
Tin cans: 0
Tin cans: 0

real	0m0.342s
user	0m0.234s
sys	0m0.316s
```

Back to the Goat Rodeo

- Clearly both of the synchronized versions of goat_rodeo took significantly more time than the unsynchronized (but erroneous) version
  - In general, one can create very fast code if one is not concerned with getting correct results
    - E.g., a clock implementation that always returns 10:00 is very fast, but only correct twice a day

Back to the Goat Rodeo

- Clearly both of the synchronized versions of goat_rodeo took significantly more time than the unsynchronized (but erroneous) version
- Acquiring and releasing a lock takes significantly more time than incrementing or decrementing an int
  - And goat rodeo does both 1,000,000 times

There is No Free Lunch

- When it comes to synchronization, there is no free lunch
  - There are potential “Lightweight” solutions
    - E.g. Isotach, a UVA research project in the late 1990s
  - But ultimately, you have to pay the price

There is No Free Lunch

- When it comes to synchronization, there is no free lunch
  - There are potential “Lightweight” solutions
    - E.g. Isotach, a UVA research project in the late 1990s
  - But ultimately, you have to pay the price
  - But…

Atomics

- Sometimes you can do things a bit more efficiently using *atomics*
- Atomic operations, which I've mentioned before, means "indivisible
  - Atom comes from the Greek *atomos* which means indivisible
- An atomic operation is one that occurs as an indivisible unit
  - I.e., another thread cannot observe the observation part way through

Atomics

- We made accesses to tin_cans_available atomic by using locks
- There is another way: `std::atomic` class template in the `<atomic>` header
  - Provides primitives often used in lock-free concurrent programming
  - How? On many modern architectures, the CPUs support atomic instructions
    - So you’re getting synchronization in hardware, rather than software, which can be faster

Atomics

- We'll discuss one example using atomics, but be warned: **Devising your own lock-free solution is incredibly difficult to do correctly and is best left to experts!**
- However, in some very simple situations (e.g., `goat_rodeo`) you can use `std::atomic` relatively easily

std::atomic Template Specialization for Fundamental Types

Template specialization	Alias
std::atomic<bool>	std::atomic_bool
std::atomic<char>	std::atomic_char
std::atomic<unsigned char>	std::atomic_uchar
std::atomic<short>	std::atomic_short
std::atomic<unsigned short>	std::atomic_ushort
std::atomic<int>	std::atomic_int
std::atomic<unsigned int>	std::atomic_uint
std::atomic<long>	std::atomic_long
std::atomic<unsigned long>	std::atomic_ulong
std::atomic<long long>	std::atomic_llong
std::atomic<unsigned long long>	std::atomic_ullong
std::atomic<char16_t>	std::atomic_char16_t
std::atomic<char32_t>	std::atomic_char32_t
std::atomic<wchar_t>	std::atomic_wchar_t

```cpp
#include <atomic>
#include <future>
#include <iostream>

using namespace std;

void goat_rodeo() {
  const size_t iterations{ 1'000'000 };
  atomic_int tin_cans_available{};
  auto eat_cans = async(launch::async, [&] {
    for(size_t i{}; i < iterations; i++)
      tin_cans_available--;
  });
  auto deposit_cans = async(launch::async, [&] {
    for(size_t i{}; i < iterations; i++)
      tin_cans_available++;
  });
  eat_cans.get();
  deposit_cans.get();
  cout << "Tin cans: " << tin_cans_available << "\n";
}

int main() {
  goat_rodeo();
  goat_rodeo();
  goat_rodeo();
}
```

```
(base) m1-mcs-dszajda:chapter_19 dszajda$ time ./goat_rodeo
Tin cans: 82528
Tin cans: -895833
Tin cans: 975992

real	0m0.035s
user	0m0.041s
sys	0m0.003s
(base) m1-mcs-dszajda:chapter_19 dszajda$ time ./goat_rodeo_locks
Tin cans: 0
Tin cans: 0
Tin cans: 0

real	0m0.310s
user	0m0.209s
sys	0m0.282s
(base) m1-mcs-dszajda:chapter_19 dszajda$ time ./goat_rodeo_guards
Tin cans: 0
Tin cans: 0
Tin cans: 0

real	0m0.345s
user	0m0.230s
sys	0m0.325s
(base) m1-mcs-dszajda:chapter_19 dszajda$ time ./goat_rodeo_atomic
Tin cans: 0
Tin cans: 0
Tin cans: 0

real	0m0.145s
user	0m0.265s
sys	0m0.003s
```