Functional Programming; Threads

ITP 435
Week 5, Lecture 2

Lambdas Review

Lambda Expressions – Motivation

- By default, std::sort sorts in ascending order:

```
std::vector<int> v;
v.emplace_back(10);
v.emplace_back(-10);
v.emplace_back(100);
std::sort(v.begin(), v.end());
// v = {-10, 10, 100}
```

- What if we want descending order?

A Custom Comparator!

```
std::sort(v.begin(), v.end(), [](int a, int b) {
   // Return a greater than b instead of less
   return a > b;
});
// v = {100, 10, -10}
```

- This uses a new syntax…

Lambda Expressions

- A ***lambda expression*** is an inline declaration of a function and implementation

```
[](int a, int b) {
   // Return a greater than b instead of less
   return a > b;
}
```

- In other languages, this may be called anonymous functions (like in JavaScript)
- Let's explain this crazy syntax!

Lambda Expression Syntax

Capture Clause
(empty in this case)

Function Parameters

```
[](int a, int b) {
    return a > b;
}
```

Body of
function

Lambda Expression Syntax, Another Example

Capture Clause

Function Parameters

Return Value
(optional)

Body

```
[&count](int i)    -> int
{
    std::cout << i << std::endl;
    count++;
    return count;
}
```

Capture Clause

- This is used to “capture” variables that exist outside the lambda expression, and use them inside the lambda
- Variables can be captured by value or reference:

```
// Capture ALL local variables by value (not recommended)
[=]
// Capture ALL local variables by reference (not recommended)
[&]
// Capture x by value and y by reference
[x, &y]
// Capture count by reference, and all other locals by value
[=, &count]
// Capture this by value (can’t be captured by reference)
// If you want to use any member functions or variables in the
// lambda, you have to capture this.
[this]
```

Lambdas stored in variables

- You're allowed to store lambdas in a variable
- Instead of figuring out the type, use auto
- Our previous example could be:

```
auto greater = [](int a, int b) {
    return a > b;
};
std::sort(v.begin(), v.end(), greater);
// v = {100, 10, -10}
```

Specifying the Type of a Lambda

- If you need to specify the type, use the std::function template class (in the <functional> header):

```
std::function<bool(int, int)> greater =
    [](int a, int b) {
        return a > b;
};
```

- (This says the lambda returns a bool and takes in two ints as parameters)

Event Class Example

Event Class

```
class Event
{
public:
	// Add a handler for this event
	void AddHandler(std::function<void()> handler)
	{
		mHandlers.emplace_back(handler);
	}
	// Loop over handlers and call each lambda
	void Trigger()
	{
		for (auto& f : mHandlers)
		{
			f();
		}
	}
private:
	std::vector<std::function<void()>> mHandlers;
};
```

Event Class in action – What's the output?

```cpp
std::string name = "Sanjay";

Event myEvent;
myEvent.AddHandler([]() {
   std::cout << "This is handler 1" << std::endl;
});
myEvent.AddHandler([name]() {
   std::cout << "This is handler 2: " << name << std::endl;
});

int x = 5;
myEvent.AddHandler([&x]() {
   x++;
});

std::cout << "x = " << x << std::endl;
myEvent.Trigger();
std::cout << "x = " << x << std::endl;
```

Output

```
x = 5
This is handler 1
This is handler 2: Sanjay
x = 6
```

Omitting the Return Type

- If you omit the return type, the return type is deduced from the return statements (much like how auto deduces type):

```
auto lambda = [](int x) {
   // This will return a bool
   return x == 5;
};

auto lambda2 = [](int a) {
   // This returns an int
   if (a < 0) {
      return a * -1;
   } else {
      return a;
   }
};
```

- Otherwise, it's assumed the lambda returns void

Explicit Return Type

- You can explicitly specify the return type using a special return syntax:

```
auto lambda = []() -> void {
   // This returns void
};

auto lambda2 = []() -> int {
   // This returns an int
   return 0;
};
```

Functional Programming

Functional Programming

- What is functional programming?

- Here's a great quote: "functional programming is the mustachioed hipster of programming paradigms"

Functional Programming, Cont'd

- In functional programming, functions are ***first-class citizens***

- This means functions can…
  - Be passed as arguments to other functions
  - Can be returned from other functions
  - Can be assigned to a variable or stored in a container

- ***C++ Lambda expressions satisfy these criteria!***

Functional Programming, Cont'd

- In functional programming, ***higher-order functions*** – functions that can accept other functions as arguments – are allowed

- Three very common higher-order functions:
  - **map** – Apply a function to each element in a collection, storing the result in another collection
  - **filter** – Remove elements from a collection based on a filtering function
  - **reduce** – Reduce a collection to a single value by applying a binary operation repeatedly

- All three of these higher-order functions are more or less supported in C++, though they have different names

Functional Programming, Cont'd

- Generally, functional programming tries to largely limit ***side effects*** – having any interaction with the world "outside" from the instance of a function call

- Examples of side effects:
  - Modifying data passed into the function (eg. pass-by-reference where you change the values)
  - Modifying global or static data
  - **Any I/O** (*can be hard to realistically enforce*)

Functional Programming, Other Practices

- Functional programs should generally be stateless – in a practical sense, this means no globals, statics, etc
- All functions should accept at least one argument
- All functions must return data or another function – no void functions
- Avoid loops/iteration – always use recursion or higher-order functions. Some functional programming languages do not have any iteration at all

Average a Vector – An Example

```
// Imperative-style implementation
float averageVector(const std::vector<float>& v) {
    float sum = 0.0f;
    for (auto& i : v) {
        sum += i;
    }

    return sum / v.size();
}
```

Average a Vector, Cont'd

- To follow the principles of functional programming, the loop has to go…

```
float averageVector(const std::vector<float>& v) {
    float sum = 0.0f;
    for (auto& i : v) {
        sum += i;
    }

    return sum / v.size();
}
```

Functional Decomposition

- ***Functional decomposition*** is a fancy way of saying “break a function into sub functions”

- Let’s make an average function that computes the average given a sum and a quantity:

```
float average(float sum, size_t qty) {
    return sum / qty;
}
```

Reduce

- ***reduce*** – take a collection and reduce it to a single value by applying a binary operator repeatedly

- ***We want to reduce the collection to the sum of its components!***

- In C++, reduce can be implemented via `std::accumulate` in the `<numeric>` header

Sum Vector

```
float sumVector(const std::vector<float>& v) {
   // std::accumulate can work as a REDUCE
   return std::accumulate(v.begin(), // Start of range
      v.end(), // End of range
      0.0f, // Initial value
      // Binary lambda expression
      [](const float& a, const float& b) {
         return a + b;
   });
}
```

Putting it Together

```cpp
// Functional implementation
float average(float sum, size_t qty) {
   return sum / qty;
}

float sumVector(const std::vector<float>& v) {
   return std::accumulate(v.begin(), v.end(), 0.0f);
}

float averageVector(const std::vector<float>& v) {
   return average(sumVector(v), v.size());
}
```

all_of, any_of, none_of – A lambda usage case

- A series of new functions in C++11 in `<algorithm>`

- Given a range of values and a unary predicate with this signature:

```
bool predicate(const T& a);
```

- It'll return true if ***all of***, ***any of***, or ***none of*** the elements satisfy the condition

all_of Example

```
std::vector<int> v1{ 2, 4, 6, 8, 10 };
if (std::all_of(v1.begin(), v1.end(), [](const int& i) {
    return (i % 2) == 0;
}))
{
    std::cout << "All are even!" << std::endl;
}
else
{
    std::cout << "All aren't even" << std::endl;
}
```

copy_if – Another Example

```
std::vector<int> from{ 1, 2, 3, 4, 5 };
std::vector<int> to;

auto is_odd = [](const int& i) { return i % 2 == 1 };

// Copy from the "from" container into the "to" container,
// only if is_odd returns true for that element
std::copy_if(from.begin(), from.end(),
             std::back_inserter(to), is_odd);
```

copy_n

- Copies n elements from a source collection to a back_inserted collection

Map

- The ***map*** higher-order function applies a function to each element, saving the results in a different collection
- It can be approximated by the std::transform function:

```
std::vector<float> divEachBy(const std::vector<float>& v,
                             float denominator) {
   std::vector<float> ret;
   // std::transform can be used to map
   std::transform(v.begin(), // Start of range
      v.end(), // End of range
      std::back_inserter(ret), // Collection to insert into
      // Unary Function that returns transform value
      [denominator](const float& a) {
         return a / denominator;
   });
   return ret;
}
```

Write our own "map" function?

```
template <typename T, typename U>
T map(const T& v, U f) {
    T ret;
    std::transform(v.begin(), v.end(),
                   std::back_inserter(ret), f);
    return ret;
}
```

- (Aside: This is an example of where using namespace std may not be good)

Rewriting divEachBy

- Now we can simplify the code:

```
std::vector<float> divEachBy(const std::vector<float>& v,
                             float denominator) {
   return map(v, [denominator](const float& a) {
      return a / denominator;
   });
}
```

Filter

- The ***filter*** higher-order function removes elements based on a Boolean filter condition

- We could write this higher-order function, too:

```
template <typename T, typename U>
T filter(const T& v, U f) {
   T ret;
   std::copy_if(v.begin(), // Start of range
      v.end(), // End of range
      std::back_inserter(ret), // Where to insert
      f); // Boolean unary function
   return ret;
}
```

Filter in Action

```cpp
std::list<int> filterIsPositive(const std::list<int>& l) {
   return filter(l, [](const int& a) {
      return a > 0;
   });
}
```

In-class activity

Generate

- Helper that can be used to insert elements into a collection, where elements need to be initialized with a particular function:

```
// Make a vector w/ 10 elements
std::vector<int> v(10);
std::generate(v.begin(), // start of range
   v.end(), // end of range
   // Lambda that returns generated value
   []() {
      return rand();
});
```

adjacent_difference

```
std::vector<int> from({ 10, 15, 20, 25, 30, 35 });
std::vector<int> to;
std::adjacent_difference(from.begin(), // Start of range
   from.end(), // End of range
   std::back_inserter(to), // Collection to insert into
   // Lambda to compute difference (defaults to subtraction)
   [](int a, int b) {
      return a - b;
});
```

- The first element in the resulting range is a copy, the rest of the collection is differences (index 1 – index 0 of the source is stored in index 1 of the destination, and so on)
- So in the above, “to” will contain:

```
{ 10, 5, 5, 5, 5, 5 }
```

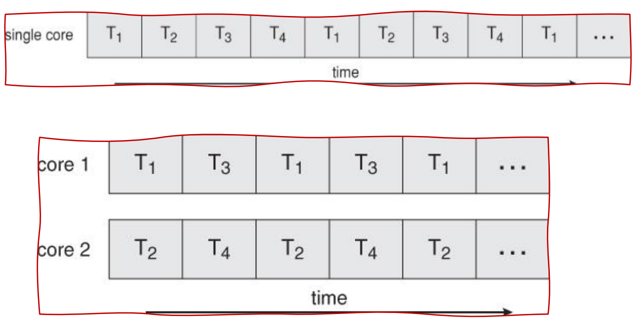
Threads Basics

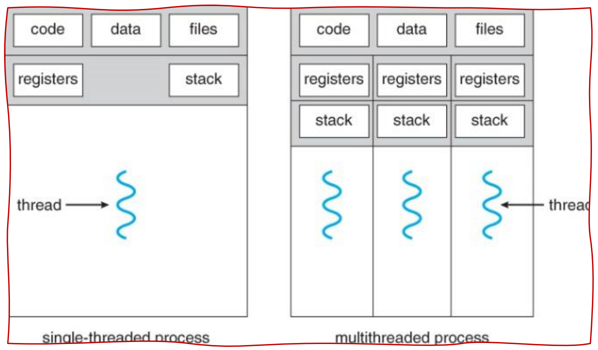
Process

- A computer program:
  - A passive collection of instructions typically stored in a file on disk.
- A process is a program in execution
  - **Allocated memory** (heap, stack, code).
  - **OS descriptors** of resources that are allocated to the process: such as file descriptors.
  - **Security attributes**, such as the process owner and the process' set of permissions.
  - **Processor state** (context), such as the content of registers and physical memory addressing.
- Several processes may be associated with the same program
  - Multiple instances of a program running at the same time.

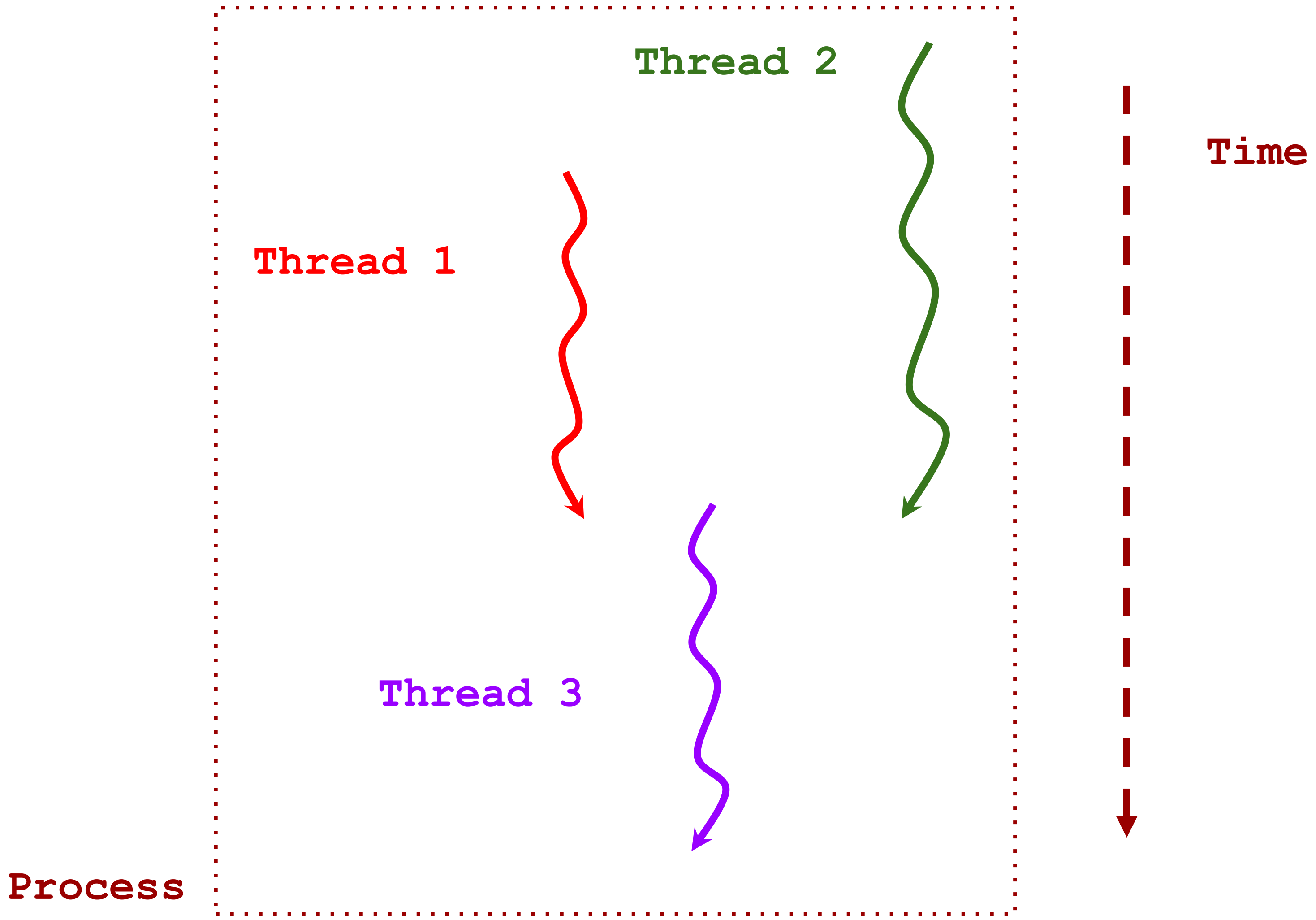
Thread

- The smallest sequence of instructions that can be managed independently by a OS.
- Multiple threads can exist within one process
  - Can execute concurrently and sharing resources such as memory, while different processes do not share these resources.
- Threads of a process share
  - Code
  - Memory (Global variables, heap)





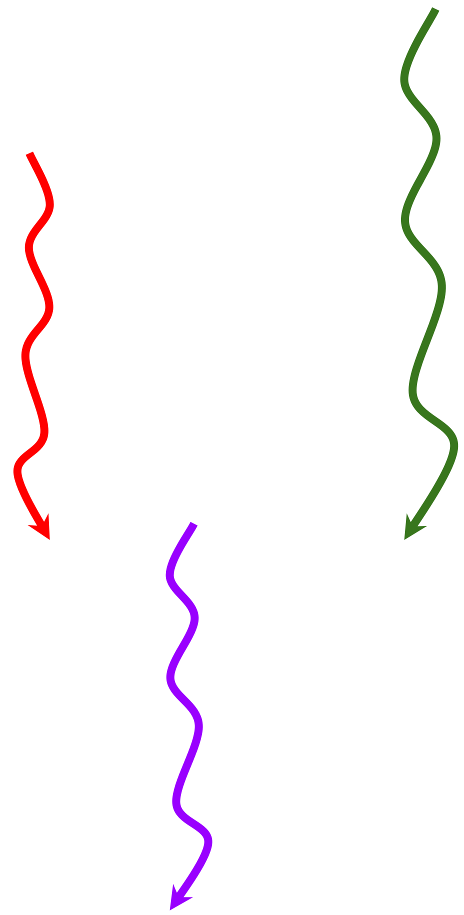
Threads



What do we cover?

What Do We Cover?

- **Threads**: Create using `std::thread`
  - Doesn't return a value
- **Tasks**: Create using `std::async`
  - Returns a value
- Both can use:
  - Pointer to function
  - Functor
  - Lambda functions



`std::thread`

Create Threads Using

Function Pointers

Calculating the sum of numbers in the range [**start**, **end**)

```
void AccumulateRange(uint64_t &sum, uint64_t start,
                         uint64_t end) {
  sum = 0;
  for (uint64_t i = start; i < end; i++) {
    sum += i;
  }
}
```

Calculating the sum of numbers in the range [**start**, **end**)

Two threads each working on half of the range

`Thread 1` `Thread 2`

0	1	2	...	...	...	...	...	...	end-1

`partial_sum[0]` `partial_sum[1]`

```
total = partial_sum[0] + partial_sum[1]
```

Pointer to Function

```
void AccumulateRange(uint64_t &sum, uint64_t start,
                     uint64_t end) {
  sum = 0;
  for (uint64_t i = start; i < end; i++) {
    sum += i;
  }
}
```

Functi on

```
std::thread t1(AccumulateRange,

               std::ref(partial_sums[0]), 0, 1000/2 );
```

Function parameters

All parameters are passed by **Value**

Use **std::ref** to pass by **reference**

Function parameters

```
std::thread t1(AccumulateRange,

               std::ref(partial_sums[0]), 0, 1000/2 );
```

Create AND Start each thread

Wait for threads to end

```
int main() {
 const int number_of_threads = 2;
 const int number_of_elements = 1000 * 1000 * 1000;
 const int step = number_of_elements / number_of_threads;
 std::vector<uint64_t> partial_sums(number_of_threads);

 std::thread t1(AccumulateRange, std::ref(partial_sums[0]), 0, step);
 std::thread t2(AccumulateRange, std::ref(partial_sums[1]), step,
               number_of_threads * step);

 t1.join();
 t2.join();

 uint64_t total =
     std::accumulate(partial_sums.begin(), partial_sums.end(), uint64_t(0));
 PrintVector(partial_sums); // Prints partial_sums
 std::cout << "total: " << total << std::endl;

 return 0;
}
```

Vector of Threads

Vector of threads

Create, start, and push each thread into a vector

Wait for threads to end

```
int main() {
 const int number_of_threads = 10;
 uint64_t number_of_elements = 1000 * 1000* 1000;
 uint64_t step = number_of_elements / number_of_threads;
 std::vector<std::thread> threads;
 std::vector<uint64_t> partial_sums(number_of_threads);
 for (uint64_t i = 0; i < number_of_threads; i++) {
   threads.push_back(std::thread(AccumulateRange, std::ref(partial_sums[i]),
                                 i * step, (i + 1) * step));
 }
 for (std::thread &t : threads) {
   if (t.joinable()) {
     t.join();
   }
 }
 uint64_t total =
     std::accumulate(partial_sums.begin(), partial_sums.end(), uint64_t(0));
 PrintVector(partial_sums);
 std::cout << "total: " << total << std::endl;

 return 0;
}
```

```
std::thread t1(AccumulateRange,
               std::ref(partial_sums[0]), 0, 1000/2 );
```

What do you need to take away?

- `std::thread` creates a **new** thread.
  - The **first parameter**: the name of the `function`.
  - The **rest of the parameters** will be passed to `function`.
  - All parameters passed to `function` are **passed by value**.
  - For pass by reference, wrap them in `std::ref`.
- **No** return value!
  - Store return value in one of the parameters passed by reference.
- Each thread **starts** as soon as it gets **created**.
- We use `join()` function to wait for a thread to finish

Create Threads Using **Functors**

Reminder: a functor is a **class/struct** that defines **operator()**

```
// comparator predicate: returns true if a < b, false otherwise
struct IntComparator
{
  bool operator()(const int &a, const int &b) const
  {
    return a < b;
  }
};

int main()
{
    std::vector<int> items { 4, 3, 1, 2 };
    std::sort(items.begin(), items.end(), IntComparator());
    return 0;
}
```

Wikipedia

Calculating the sum of numbers in the range [**start**, **end**)

```
class AccumulateFunctor {
public:
 void operator()(uint64_t start, uint64_t end) {
   _sum = 0;
   for (auto i = start; i < end; i++) {
     _sum += i;
   }
   std::cout << _sum << std::endl;
 }
 uint64_t _sum;
};
```

Vector of threads

Vector of functors

Create, start, and push each thread into a vector

Wait for threads to end

Calculate total sum

```
const int number_of_threads = 10;
uint64_t number_of_elements = 1000 * 1000 * 1000;
uint64_t step = number_of_elements / number_of_threads;
std::vector<std::thread> threads;
std::vector<AccumulateFunctor *> functors;
for (int i = 0; i < number_of_threads; i++) {
  AccumulateFunctor *functor = new AccumulateFunctor();
  threads.push_back(
      std::thread(std::ref(*functor), i * step, (i + 1) * step));
  functors.push_back(functor);
}
for (std::thread &t : threads) {
  if (t.joinable()) {
    t.join();
  }
}
int64_t total = 0;
for (auto pf : functors) {
  total += pf->_sum;
}
std::cout << "total: " << total << std::endl;
```

```
AccumulateFunctor *functor = new AccumulateFunctor();
std::thread(std::ref(*functor), 0, 1000/2));
```

What do you need to take away?

- Creates the first parameter is either:
  - The **functor object**
    - If you don’t need to use member variables later
  - a **reference to the functor object**.
    - If you need to store in and use the member variables later
- The rest of the parameters are passed to the **operator()** function.
- Return value can be **stored** in a **member variable**
  - As an alternative to passing a reference

Create Threads Using **Lambda Functions**

Reminder: Lambda function is a function definition that is not bound to an identifier.

```
[capture](parameters) -> return_type { function_body }
```

```
[](int x, int y) -> int { return x + y; }
```

```
std::vector<int> some_list{ 1, 2, 3, 4, 5 };
int total = 0;
std::for_each(begin(some_list), end(some_list),
              [&total](int x) { total += x; });
```

Wikipedia

Calculating the sum of numbers in the range [**start**, **end**)

```
[i, &partial_sums, step] {
      for (uint64_t j = i * step; j < (i + 1) * step; j++) {
        partial_sums[i] += j;
      }
    }
```

```
for (uint64_t i = 0; i < number_of_threads; i++) {
   threads.push_back(std::thread([i, &partial_sums, step] {
     for (uint64_t j = i * step; j < (i + 1) * step; j++) {
       partial_sums[i] += j;
     }
   }));
 }
```

Vector of threads

Vector of sums

Create, start, and push each thread into a vector

Wait for threads to end

Calculate total sum

```
const int number_of_threads = 10;
uint64_t number_of_elements = 1000 * 1000 * 1000;
uint64_t step = number_of_elements / number_of_threads;
std::vector<std::thread> threads;
std::vector<uint64_t> partial_sums(number_of_threads);
for (uint64_t i = 0; i < number_of_threads; i++) {
  threads.push_back(std::thread([i, &partial_sums, step] {
    for (uint64_t j = i * step; j < (i + 1) * step; j++) {
      partial_sums[i] += j;
    }
  }));
}
for (std::thread &t : threads) {
  if (t.joinable()) {
    t.join();
  }
}
uint64_t total =
    std::accumulate(partial_sums.begin(), partial_sums.end(), uint64_t(0));
PrintVector(partial_sums);
std::cout << "total: " << total << std::endl;
```

```
std::thread([i, &partial_sums, step] {
  for (uint64_t j = i * step; j < (i + 1) * step; j++) {
    partial_sums[i] += j;
  }
});
```

What do you need to take away?

- As an alternative to passing a parameter, we can pass references to lambda functions using **lambda capture**.

`std::async`

Task, Futures, and Promises

Pointer to Function

```
void AccumulateRange(uint64_t &sum, uint64_t start,
                         uint64_t end) {
  sum = 0;
  for (uint64_t i = start; i < end; i++) {
    sum += i;
  }
}
```

Functi
on

```
std::thread t1(AccumulateRange,

               std::ref(partial_sums[0]), 0, 1000/2 );
```

```
t1.join()
```

Function
parameters

Pointer to Function

```
uint64_t GetRangeSum(uint64_t start, uint64_t end) {
 uint64_t sum = 0;
 for (uint64_t i = start; i < end; i++) {
   sum += i;
 }
 return sum;
}
```

Functi
on

Return value

```
auto t = std::async(GetRangeSum, 0, 100/2)
```

Future

Function
parameters

```
return_value = t.get()
```

Vector of **tasks**

```
const int number_of_threads = 10;
uint64_t number_of_elements = 1000 * 1000 * 1000;
uint64_t step = number_of_elements / number_of_threads;
std::vector<std::future<uint64_t>> tasks;
```

Create, start, and push each **task** into a vector

```
for (uint64_t i = 0; i < number_of_threads; i++) {
  tasks.push_back(std::async(GetRangeSum, i * step, (i + 1) * step));
}
```

Wait for **tasks** to end and read **return values**

```
uint64_t total = 0;
for (auto &t : tasks) {
  total += t.get();
}

std::cout << "total: " << total << std::endl;
```

std::launch policy

- The **std::async** can take one more parameter of type **std::launch**. It can take the following values:
  - std::launch::**async**: the function will run on its own (new) thread.
  - std::launch::**deferred:** that the function call will be deferred until either wait() or get() is called on the future.
  - std::launch::**async | **std::launch::**deferred:** that the implementation may choose. (**Default option**)

```
auto t = std::async(policy, GetRangeSum, 0, 100/2)
```

How many threads?

```
unsigned int n =
std::thread::hardware_concurrency();
std::cout << n << " concurrent threads are
supported.\n";
```

Returns the number of concurrent threads supported by the implementation. The value should be considered only a hint.

```
auto t = std::async(GetRangeSum, 0, 100/2)
```

What do you need to take away?

- **Tasks** are created using **`std::async`**.
- **Future**: The **returned value** from std::async
  - We get Future's value by calling **`get()`**
- If the future values are not ready, the main thread **blocks** until the future value becomes ready (similar to **`join()`**).
- **Promise**: Return value of the function passed to **`std::async`**.
  - For the most part, you don't need to know details of std::promise or define any variable of type **std::promise**. The C++ library does that behind the scenes.
- Tasks start based on the given **std::launch** policy.

Summary of Threads/Async

Summary

- **Threads**: Create using **`std::thread`**
  - Doesn't return a value
- **Tasks**: Create using **`std::async`**
  - Returns a value
- Both can use:
  - Pointer to function
  - Functor
  - Lambda functions