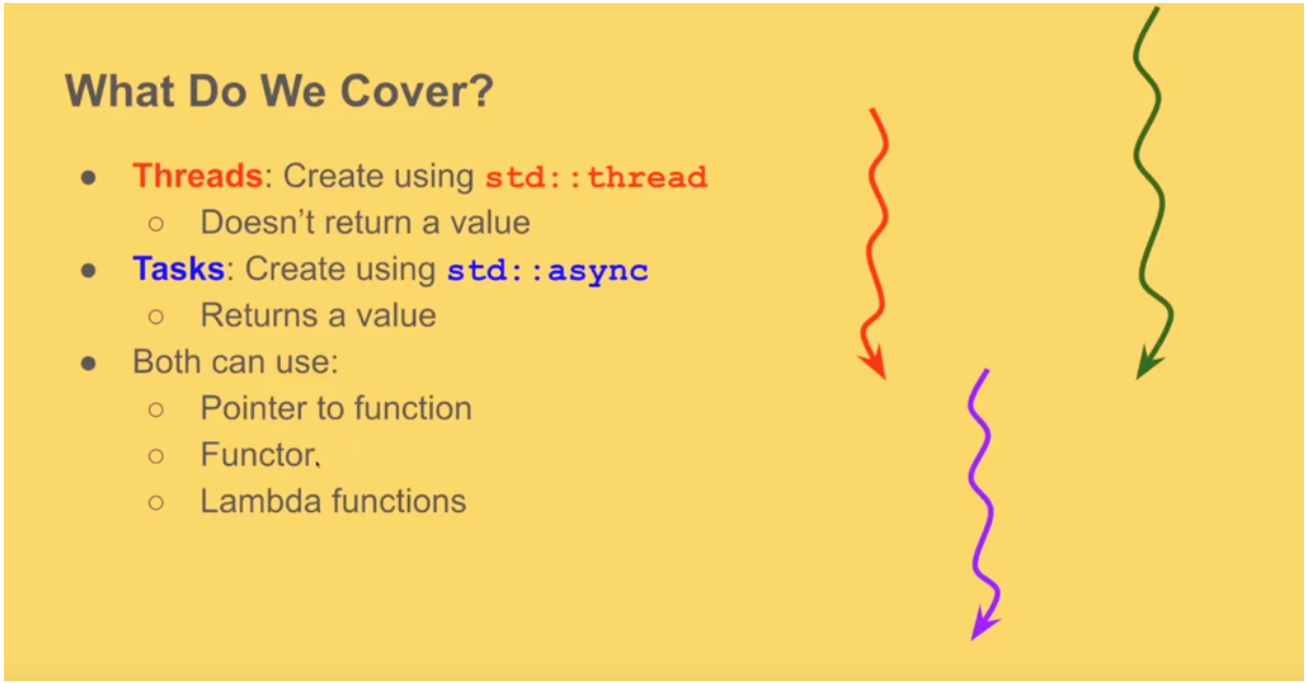
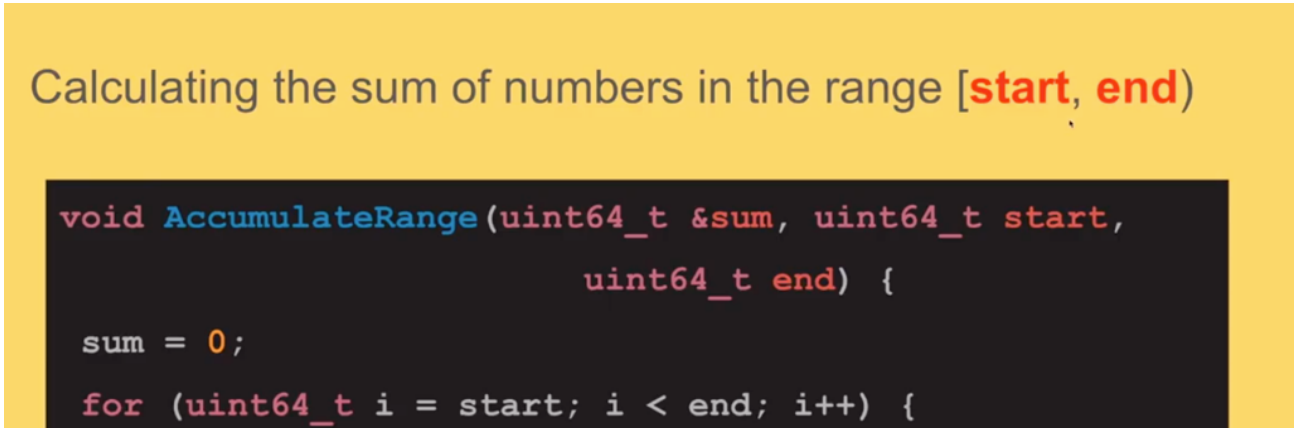
Multi-threading

Friday, March 27, 2020 17:02



Std::thread create threads using function pointers.



```
    sum += i;
  }
}
```

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

Two threads each working on half of the range


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

Pointer to Function

this first parameter used as construction.

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

Function

```
std::thread t1(AccumulateRange,
```

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

passed from the above function

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 );
```

```
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 =
```

Create AND Start each thread

Wait for threads to end

> 2 threads

> wait for finishing execution

```
    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:

```
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;
}
```

Vector of threads

Create, start, and push each thread into a vector

Wait for threads to end

no matter how many threads, the total value must be the same.

```
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, not functions:

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;
```

// create an object of this functor

// use sum member variable to get total.

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

What do you need to take away?

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

- Another method to create threads is to use Lambda function

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; });
```

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;
    }
  }));
}
```

Whole code:

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<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**.

There is one more way we can create threads.

Std::async Taks, Futures and promises:

Pointer to Function

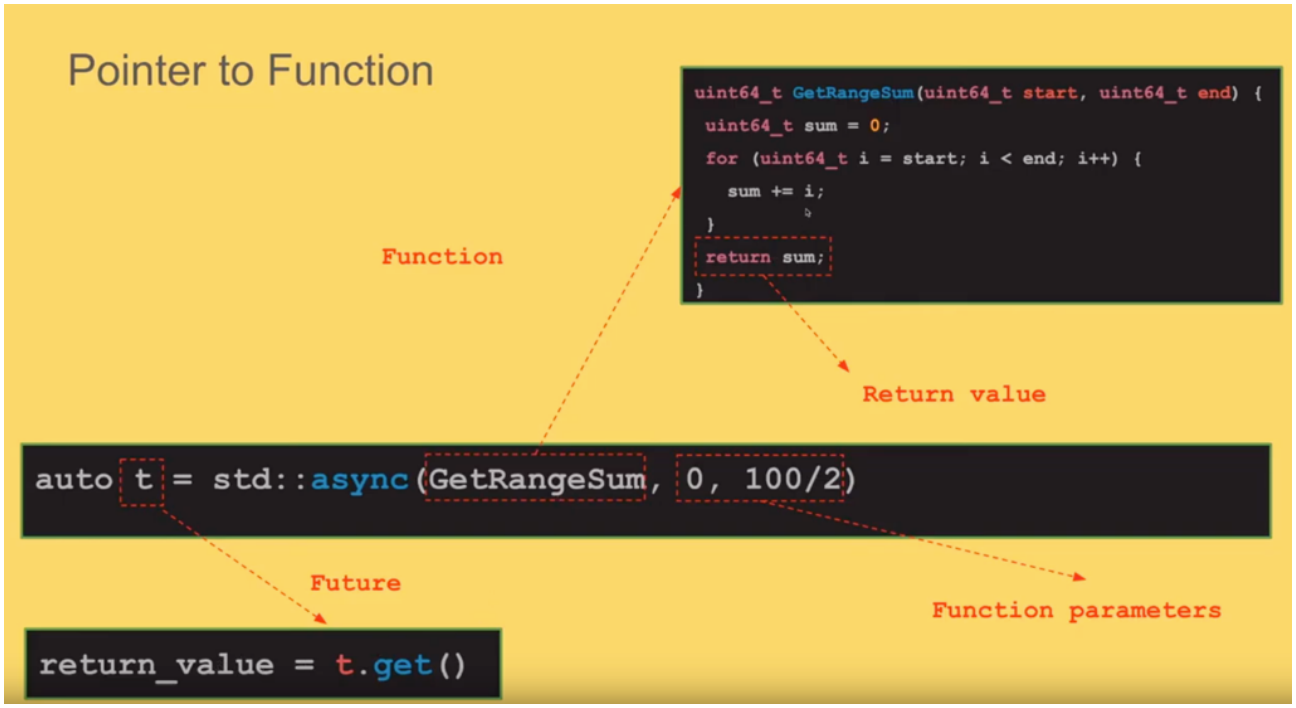
```
void AccumulateRange(uint64_t &sum, uint64_t start,
                     uint64_t end) {
 sum = 0;
 for (uint64_t i = start; i < end; i++) {
   sum += i;
 }
}
```

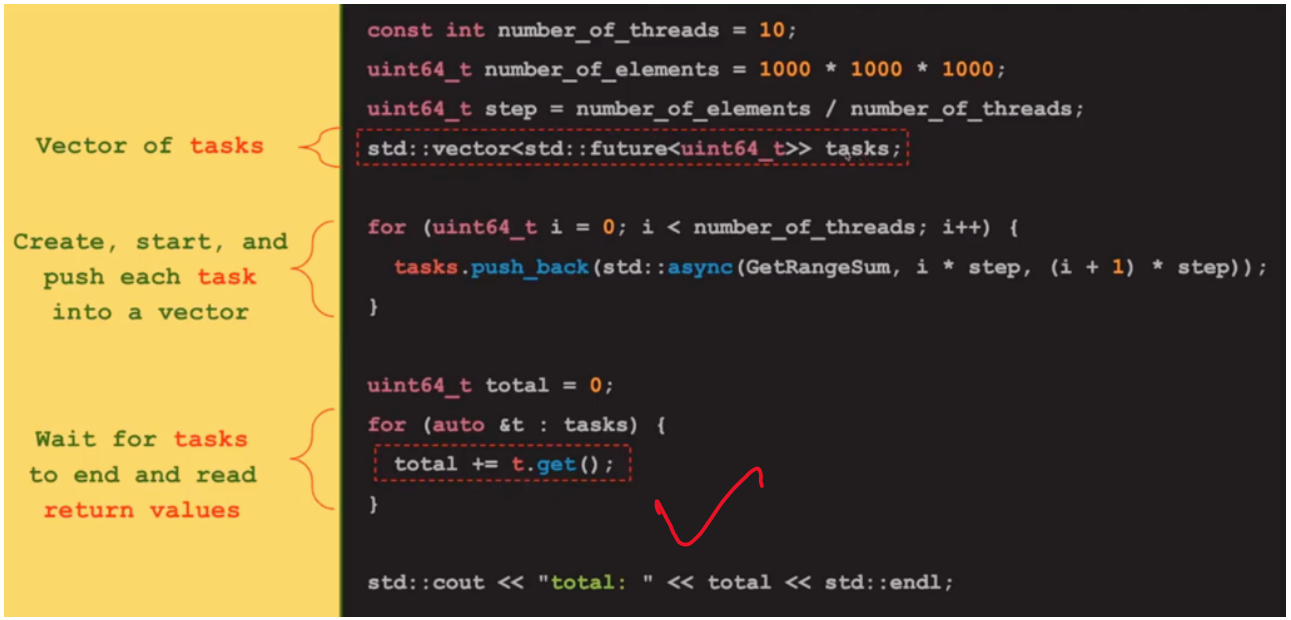
Function

```
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;
}
Function
Return value
auto t = std::async(GetRangeSum, 0, 100/2)
Future
Function parameters
return_value = t.get()


Vector of tasks
Create, start, and push each task into a vector
Wait for tasks to end and read return values
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;
for (uint64_t i = 0; i < number_of_threads; i++) {
tasks.push_back(std::async(GetRangeSum, i * step, (i + 1) * step));
}
uint64_t total = 0;
for (auto &t : tasks) {
total += t.get();
}
std::cout << "total: " << total << std::endl;

Homework

Rewrite the previous code to create tasks using:

- Functors
- Lambda Functions