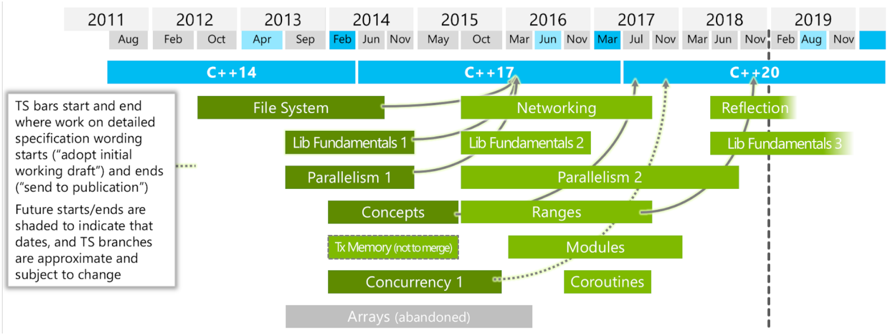
Introduction to C++20

Raleigh Littles

December 17, 2018

Overview

Current timeline



C++ committee works asynchronously

- C++ committee votes on certain features, received from proposals
- These features, once voted on, go into the C++ standard
- Particularly large/challenging modules (?) are deferred to be *technical standards*
- Technical standards are worked on independently and can be delivered later (even once the standard itself is finished!)

Current commitee progress

Meeting #1: July 2017

Voted in: Concepts, designated initializers, new lambda capture syntax, template parameter lists, ...

Meeting #2: November 2017

Voted in: variable initialization on ranged-for statements, bit-casting object representations, three-way comparison operator, atomic shared pointers, ...

Meeting #3: June 2018

Voted in: Contracts, feature test macros, Ranges, explicit booleans, `constexpr` virtual functions

Meeting #4: November 2018

Voted in: `consteval` keyword, `constexpr` try-catch, `constexpr` dynamic casts, `pointer_traits`, ...

Remaining progress

Meeting #5: Q1 2019

Last meeting to vote on new features. Begin drafting C++20 standard.

Meeting #6: Q2 2019

C++20 internal draft completed, committee begins vote to approve.

Concepts: Introduction

- "A description of supported operations on a type"
- Also known as *constraints* – can you guess why?

```
template <class Type>
concept bool EqualityComparable()
{
    return requires(Type a, Type b)
    {
            {a == b} -> Boolean;
            {a != b} -> Boolean;
    };
}
```

Concepts continued

```
template <class Type>
concept bool GenericConcept()
{
    return requires(Type a, Type b)
    {
            {OPERATION} -> TYPE;
    };
}
```

- *Does the operation make sense on this template type? Are the results convertible to a type that satisfies what you told me to expect?*
- *EqualityComparable* (previous slide) is a *concept*, as well as *TriviallyCopyable* or *ReversibleContainer*

Concepts: Benefits

- Allows you to specify *interoperability* between templates, à la template interfaces ..
- .. whereas interfaces are run-time polymorphism, concepts provide **automatic compile-time** polymorphism
- Introducing type-checking to template programming makes it *simpler*

Ranges

- Adds range comprehensions, function composition, and lazy evaluation
- Existing STL containers are being upgraded to ranges

```
std::vector<int> vector{1,2,3};

// C++11
std::sort(vector.begin(), vector.end());

// C++20
std::sort(vector);
```

Don't let the above example betray you

Ranges: function composition example

Here's what it looks like in two languages that got Ranges right.

Problem: Find all square numbers under 1000 that are also odd.

```
# Ruby
(1..1000).map{|n| n * n}.select{|i| i.odd? && (1000  > i)}

--Haskell
takeWhile(<1000) . filter odd . map(^2)
```

1,9,25,49,81,121,169,225,289,361,441,529,625,729,841,961

Ranges: function composition example

```
#include <ranges>

using namespace std;

auto odds = ranges::view::transform(
      [](int i){return i*i; } ) |
      [](int i){return i % 2 == 0; } ) |
      [](int i){return i < 1000; } );

auto oddNumbers = ranges::view::ints(1) | odds;
```

Where have we seen this kind of syntax?

Perfect squares example explained

- The pipe "|" character allows us to *compose* functions on a range
- View adapters give us a "peek" into a subsection of a range and define the iteration behavior. Don't confuse this with `string_view`!
- Our C++ example used lazy evaluation and we didn't even notice, which is good

Perfect squares example review

- `odds` is an example of a view adapter - specifically the *transform* one (others include *filter*)
- `oddNumbers` takes an infinite sequence of integers (starting with 1) and applies our view adapter to it
- To print the contents, use any for loop you like, or the new `ranges::for_each` for loop

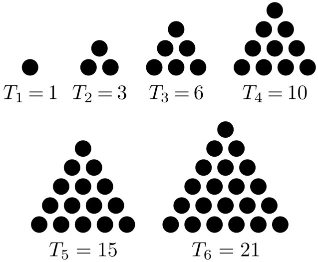
```
#include <ranges>

using namespace std;

auto odds = ranges::view::transform(
    [](int i){return i*i; } ) |
    [](int i){return i % 2 == 0; } ) |
    [](int i){return i < 1000; } );

auto oddNumbers = ranges::view::ints(1) | odds;
```

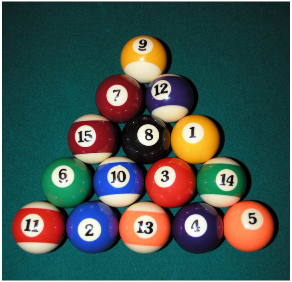
Range comprehensions

- To demonstrate these, I'll use an even more contrived example
- Confession: If you can recognize the logo below, you're already probably very familiar with this concept

Light digression: Triangular numbers

Counts the number of objects arranged in an equaliteral triangle





Heavy digression: Handshake problem

Problem

In a room of n people, how many ways are there for them to shake hands without shaking the same hand twice?

Solution

Big surprise, the answer is triangular numbers (specifically T_{n+1} in this case)

How would we do this in C++20?

Last digression: flashback to 8th grade

Rather than computing solutions directly, we can check if a number is a solution by seeing if it is a triangular number.

$$T_n = \frac{n(n+1)}{2} \qquad \text{(Formula for finding triangular numbers)}$$

$$n = \frac{\sqrt{8T_n + 1} - 1}{2} \qquad \text{(Inverse of the formula above)}$$

Result: For any integer x, if $8x + 1$ is a perfect square, then x is a solution to our problem.

Range comprehensions

Our official solution to this problem looks like this.

```
#include <ranges>
#include <cmath>

using namespace std;

auto solution = ranges::view::ints(1) |
ranges::view::for_each([](int i)
{

return yield_if( sqrt(8*i + 1) * sqrt(8*i + 1) == 8*i + 1);

});
// print the first 100 solutions
ranges::for_each(solution |
view::take_while([](int i){ return i < 100; }), [](int i)
{
     cout << i << endl;
});
```

Range comprehension in-depth

- Range comprehensions are analogous to function composition, but can be more terse (this is either a good thing or a bad thing)
- Again, we used lazy evaluation
- The yield statement was an example of a generator, which is a special case of coroutines
- Note: Commitee is still considering adding support for coroutines

Contracts

Some basic review:

- Preconditions: " a predicate that is supposed to hold upon entry in a function"
- Postconditions: " a predicate that is supposed to hold upon exit from the function"

Saying that Contracts are like `assert()` on steroids is a major understatement

Simplest example: Queues

```
auto push(Queue& queue, int value)
[[ expects: queue.full() == false ]]
[[ ensures: queue.empty() == false ]]
{
 // code goes here
 [[ assert: queue.is_ok() ]]
}
```

- Surprisingly, this is pretty self-explanatory

Contracts: in-depth

The syntax is
`[[ contract-attribute modifier: conditional expression ]]`

- `contract-attribute`: This is one of *expects*, *ensures* , *assert*
- `modifier`: Possible values are
  1. `default`
  2. `audit`
  3. `axiom`

The hype around Contracts is that *you* get to choose what each of those modifier levels means **and** what happens when your contract is violated

"Build levels"

This is a compiler-specified option that determines *what contracts will be checked*.

- `off` : No contracts whatsoever are run.
- `default` : Only contracts with the `default` level are checked
- `audit` : Contracts with the `audit` and `default` levels are checked

You cannot change the build level in source code

So what if I violate the contract?

By default, failing contracts mean program termination

- However, there are **contract violation handlers**
- As the name implies, this is the function that gets called when a contract is violated
- There is one contract violation handler per *translation unit*
  Must have specific signature:

```
void(const std::contract_violation &);
```

So is that it?

If only...

- After the contract violation handler is reached, there's two options:
  1. Program terminates indefinitely (default option is none is specified!)
  2. Program continues
- This option is called the *continuation mode* and can only be set through the compiler (not source code)
- You cannot determine the current continuation mode inside of source code

Contracts: Benefits

How does this compare to the regular `assert()` macro?

- This isn't a macro
- You can handle custom behavior on violations
- Contracts enable compilers to perform more optimizations
- With `assert()` only, its hard to impose precondition checks because you may not always have access to calling site of a function

Herb Sutter (chair of the C++ Committee) has personally said he thinks *"contracts is the most impactful feature of C++20 so far"*

Any sci-fi fans out there?

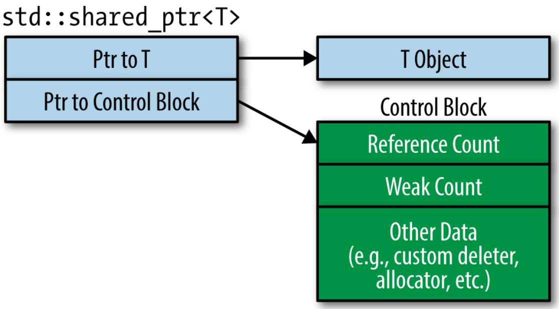
- C++20 adds a three-way comparison operator (<=>)
- Sometimes people call this the spaceship operator
- Given two values A and B, this operator checks each of the following
  1. Is $A < B$?
  2. Is $A = B$?
  3. Is $A > B$?
- Perl, Haskell and Ruby, have this
- This is an extension of C's `strcmp()`

Atomic shared pointers

- Nothing to do with chemistry
- .. Despite being the "A" in *ACID*
- "Aren't smart pointers already thread-safe?"

The answer is: kind of?

Here's how smart pointers are implemented



- The control block is thread-safe
- Access to the object *isn't*

Wanna race?

```cpp
    std::shared_ptr<int> pointer  = std::make_shared<int>(2011);

    for (auto i = 0; i < 10; i++)
    {
        std::thread([&pointer]
        {
                pointer = std::make_shared<int>(2014);
        }).detach();

    }
```

I can't take credit for this example, but it illustrates the point...

To be fair...

- There technically are atomic operations you can do on shared pointers
- `std::atomic_is_lock_free`, `std::atomic_exchange`, a handful of others
- Like with many things in C++, the right usage of these takes discipline
- ... which make it error-prone
- Atomic pointers are going to solve this

Don't believe me?

```cpp
template<typename Type> class concurrent_stack {
    struct Node { Type t; shared_ptr<Node> next; };
    atomic_shared_ptr<Node> head;
    concurrent_stack( concurrent_stack &) =delete;
    void operator=(concurrent_stack&) = delete;

    public:
    concurrent_stack() =default;
    ~concurrent_stack() =default;
    class reference {
        shared_ptr<Node> p;
    public:
        reference(shared_ptr<Node> p_) : p{p_} { }
        Type& operator* () { return p->t; }
        Type* operator->() { return &p->t; }
    };

    auto find(Type t) const {
        auto p = head.load();
        while( p && p->t != t )
            p = p->next;
        return reference(move(p));
    }
    auto front() const {
      return reference(head);
    }
    void push_front(Type t) {
      auto p = make_shared<Node>();
      p->t = t;
      p->next = head;
      while( !head.compare_exchange_weak(p->next, p) ){ }
    }
    void pop_front() {
      auto p = head.load();
      while( p && !head.compare_exchange_weak(p, p->next) ){ }
    }
};
```

Told you

That last slide:

- was a **40-line** implementation of a *thread-safe*, (singly) linked list
- The word "atomic" only showed up once
- You could have written thread-safe code without knowing what a thread was
- In C++17, you'd have had to use the atomic member functions every time you touched the head pointer

The End