C++20 Concepts

A Day in the Life

Saar Raz • 2019

About me

- Saar Raz, 25 from Israel
  - C++ enthusiast in my spare time
  - Also like graphic design, video games
- Implementer of C++20 Concepts in the Clang compiler
  - (Come to my other talk Wednesday 1515 to hear that story)

This Talk

- Concepts is a new feature in C++20 (next year!)
- Concepts is a metaprogramming feature
  - More ways to write generic code
  - Should you care if you are not writing a generic library?
    - YES

Thank You!

Saar Raz • saar@raz.email

Concepts – Why Care?

- Performance
- Metaprogramming can help us achieve optimal performance
- You may not use metaprogramming every day
  1. Hard to write and debug
  2. Hard to read and reason about
  3. Hard to compile
- Then why care about another metaprogramming feature?
  - Because!
    1. Concepts make metaprogramming easier to write and debug
    2. Concepts make metaprogramming easier to read and reason about
    3. Concepts make metaprogramming easier to compile
- Concepts – metaprogramming for everyone

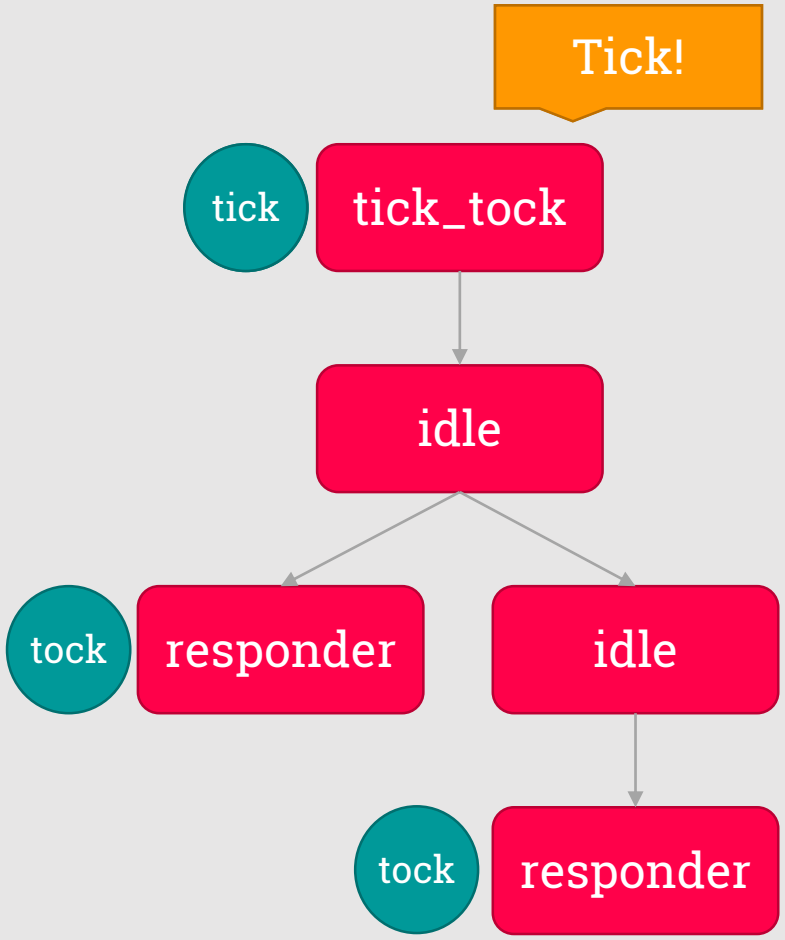
Today

- We'll use concepts for a specific metaprogramming task
  - We'll see that it's not that scary now!
  - We'll see cool things that happen in a C++20 world with concepts
- The task is specific, but the idea is not:
  - You might have an idea how to use metaprogramming to improve your project
  - It is not so hard to do now with C++20
  - Go do it!

The Mission

- We want to write the base for a game engine
- More specifically, we want a component-messaging system
  - World is built out of a hierarchy of components
  - Components send and respond to messages from one another

Simple component tree



STRAIGHTFORWARD IMPLEMENTATION

- We can easily implement this using virtual calls and RTTI:

```
struct component_base {
  virtual void handle(const message_base &m) = 0;

  vector<unique_ptr<component_base>> children;
  component_base* parent;

  void sendDown(message_base &message) {
    for (unique_ptr<component_base> &c : children) {
      c->handle(message);
      c->sendDown(message);
    }
  }

  void sendUp(message_base &message) {
    if (!parent) return;
    parent->handle(message);
    parent->sendUp(message);
  }
};
```

NICER IMPLEMENTATION

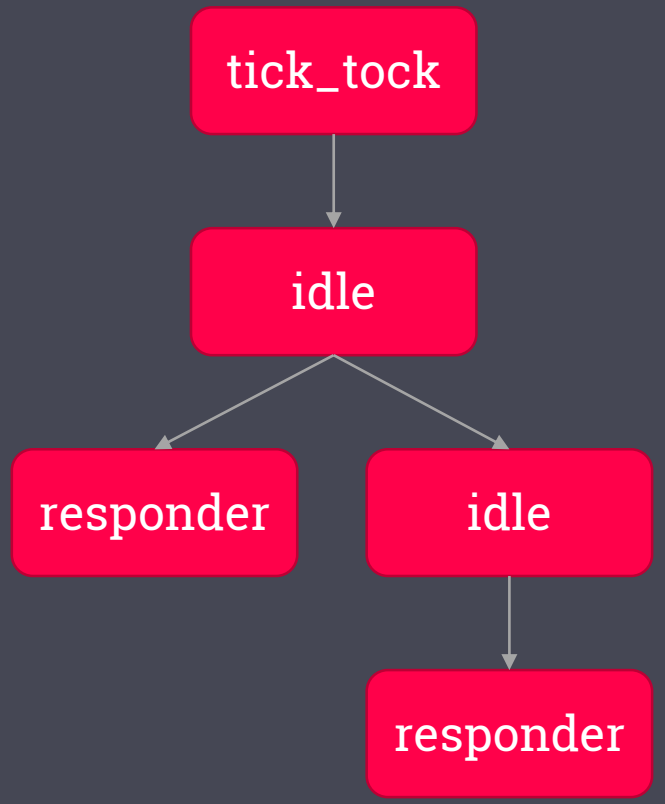
- With a little more work we can get:

```
template<typename... Ms>
struct component // ...

struct start{}; struct tick{}; struct tock{};

struct tick_tock: component<start, tock> {
  void handle(const start &message) override {
    puts(“tick“);
    sendDown(tick{});
  }
  void handle(const tock &message) override {
    puts(“tock”);
  }
};

struct responder: component<tick> {
  void handle(const tick &message) override {
    sendUp(tock{});
  }
};
```



Compile time!

How does the assembly look?

Static idea

- A child cannot know its parents at compile time!
  - We'd have a cyclic dependency
- Alternative approach:
  - Have the entire tree known beforehand
  - Pass a reference to it to handler functions

DETACHED TREE MODEL

```
using component_tree = tree<unique_ptr<component_base>>;

struct tick_tock: component<start, tock> {

  void handle(const start &message, component_tree& tree,
              component_tree::iterator location) override {
    puts(“tick“);
    sendDown(tree, location, tick{});
  }
  ...
};

struct responder: component<tick> {

  void handle(const tick &message, component_tree& tree,
              component_tree::iterator location) override {
    sendUp(tree, location, tick{});
  }
};
```

PACKAGING TREE AND LOCATION

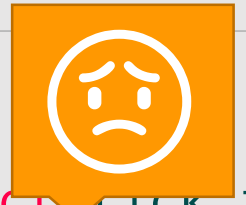
```
struct tick_tock: component<start, tock> {

  void handle(const start &message, context context) override
  {
    puts(“tick“);
    context.sendDown(tick{});
  }
  ...
};

struct responder: component<tick> {

  void handle(const tick &message, context context) override
  {
    context.sendUp(tock{});
  }
};
```

MAKING IT STATIC



```
struct tick_tock: component<start, tock> {
  template<typename Context>
  void handle(const start &message, Context context) override
  {
    puts("tick");
    context.sendDown(tick{});
  }
  ...
};

struct responder: component<tick> {
  template<typename Context>
  void handle(const tick &message, Context context) override
  {
    context.sendUp(tock{});
  }
};
```

Concepts to the rescue!

Let’s see how we can make this better…

Compile-time static context

- context = tree and a ‘location’ in the tree
  - We therefore need compile-time static tree and tree locations
- Anyway, it’s going to be a template, something like

```
template<typename Tree_, typename TreeLocation_>
struct context {
  Tree_& tree;
  TreeLocation_ location;

  void sendDown(...) { ... }
  void sendUp(...) { ... }
};
```

- Maybe we can use this to define the Context concept?

Different ways to define a concept

```cpp
template<typename Root_, typename... Children_>
struct tree {
  Root_ root;
  tuple<Children_...> children;
};
template<typename T>
concept Tree = is_specialization_of_v<T, tree>;
```

- Most accurate
- Requires some template metaprogramming
- Not easily extensible
- Can’t use concept in the class itself

Different ways to define a concept

```
struct tree_base {};

template<typename Root_, typename... Children_>
struct tree: tree_base {
  Root_ root;
  tuple<Children_...> children;
};
template<typename T>
concept Tree = std::derived_from<T, tree_base>;
```

- Easiest
- Fastest
- Innacurate
- “Hack”

Different ways to define a concept

```
template<typename Root_, typename... Children_>
struct tree {
  Root_ root;
  tuple<Children_...> children;
};
template<typename T>
concept Tree = requires (T t) {
  { t.root } -> Node;
  …
};
```

- Extensible
- Can use concept in class
- DRY?
- Performance?
- Harder to write correctly → tests!

Moving on!

We'll skip over tree_location for now...

C++20 Concepts

A Day in the Life

Saar Raz • 2019 • saar@raz.email • @saarraz1

Talk to JetBrains people for questions about CLion integration

Try it out: concepts.godbolt.org

Build Clang with concepts: github.com/saarraz/clang-concepts-monorepo