C++20: Concepts

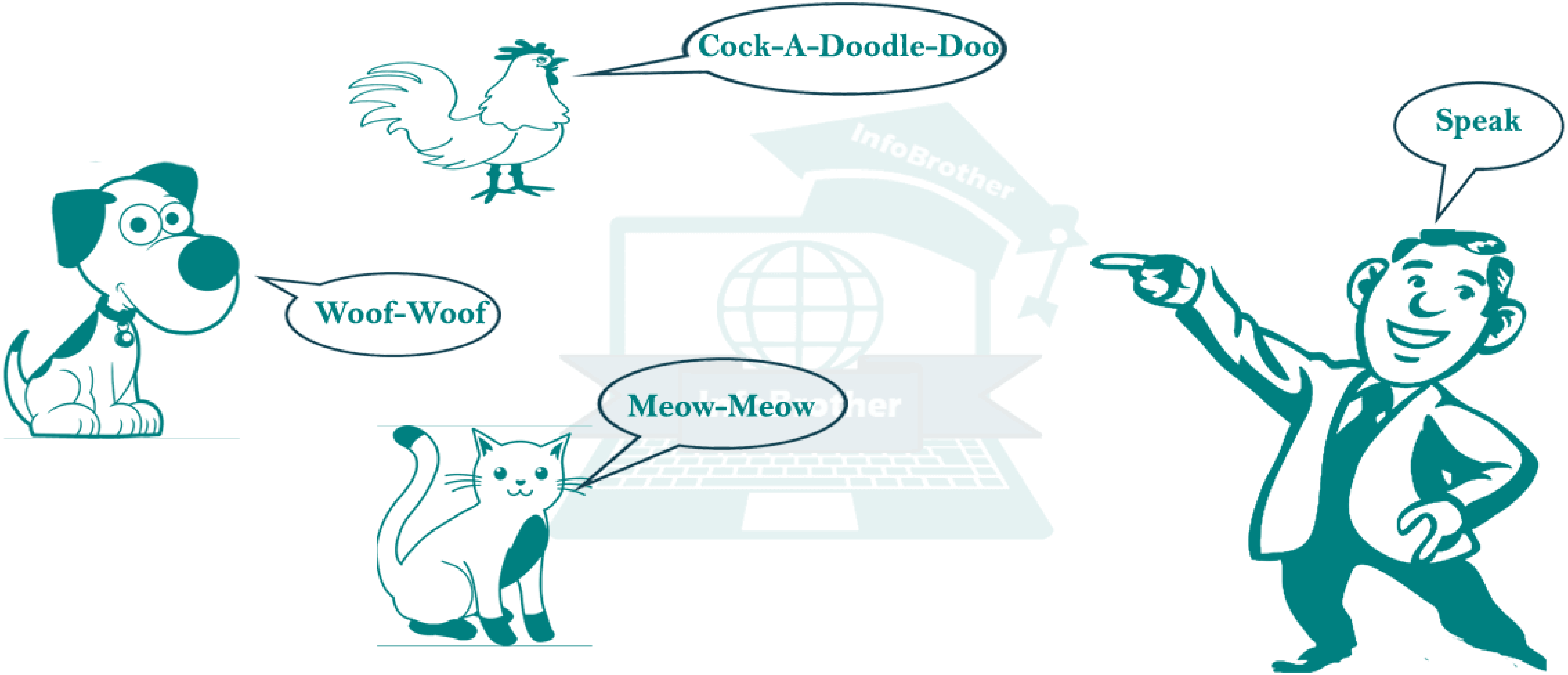
Serbian C++ User Group meetup,
Belgrade, August 2022, ICT Hub

Goran Aranđelović
goran.arandjelovic@gmail.com
cppserbia@cpplang.com

Concepts are an extension to the templates feature provided by the C++ programming language. Concepts are named Boolean predicates on template parameters, evaluated at compile time. A concept may be associated with a template (class template, function template, member function of a class template, variable template, or alias template), in which case it serves as a constraint: it limits the set of arguments that are accepted as template parameters.

— Wikipedia, August 2022

Polymorphism


Cock-A-Doodle-Doo
Speak
Woof-Woof
Meow-Meow

Polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

— Wikipedia, August 2022

Polymorphism is the provision of a single interface to entities of different types or <u>the use of a single symbol to represent multiple different types</u>.

— Wikipedia, August 2022

Polymorphism

Types:

- Ad-hoc polymorphism
- Parametric polymorphism
- Subtyping
- Row polymorphism
- Polytypism

Implementation aspects:

- Static polymorphism
- Dynamic polymorphism

Ad-hoc polymorphism

Function overloading:

```
void print_me(int i) #1
{
  // ...
}
void print_me(double d) #2
{
  // ...
}
void print_me(std::string s) #3
{
  // ...
}

print_me(17.0);          // calls #2
print_me(8);             // calls #1
print_me("2022"s);       // calls #3
```

Parametric polymorphism

Function template:

```cpp
struct point { int x; int y; int z; };

template<typename T>
std::string serialize(T object)
{
  // ...
}

auto p = point{1, 2, 3};
auto v = std::vector<point>{{4, 5, 6}, {7, 8, 9}};

auto result1 = serialize(1729);
auto result2 = serialize(p);
auto result3 = serialize(v);
```

Subtyping

Inheritance and virtual functions:

```
struct animal {
  virtual void speak() = 0;
};

struct dog : animal {
  virtual void speak() override { /* ... */ }
}

struct cat : animal {
  virtual void speak() override { /* ... */ }
}

void say_something(animal *a)
{
  a->speak();
}
```

Ad-hoc polymorphism

Function overloading:

```cpp
void print_me(int i) #1
{
  // ...
}
void print_me(double d) #2
{
  // ...
}
void print_me(std::string s) #3
{
  // ...
}

print_me(17.0);        // calls #2
print_me(8);           // calls #1
print_me("2022"s);     // calls #3
```

Parametric polymorphism

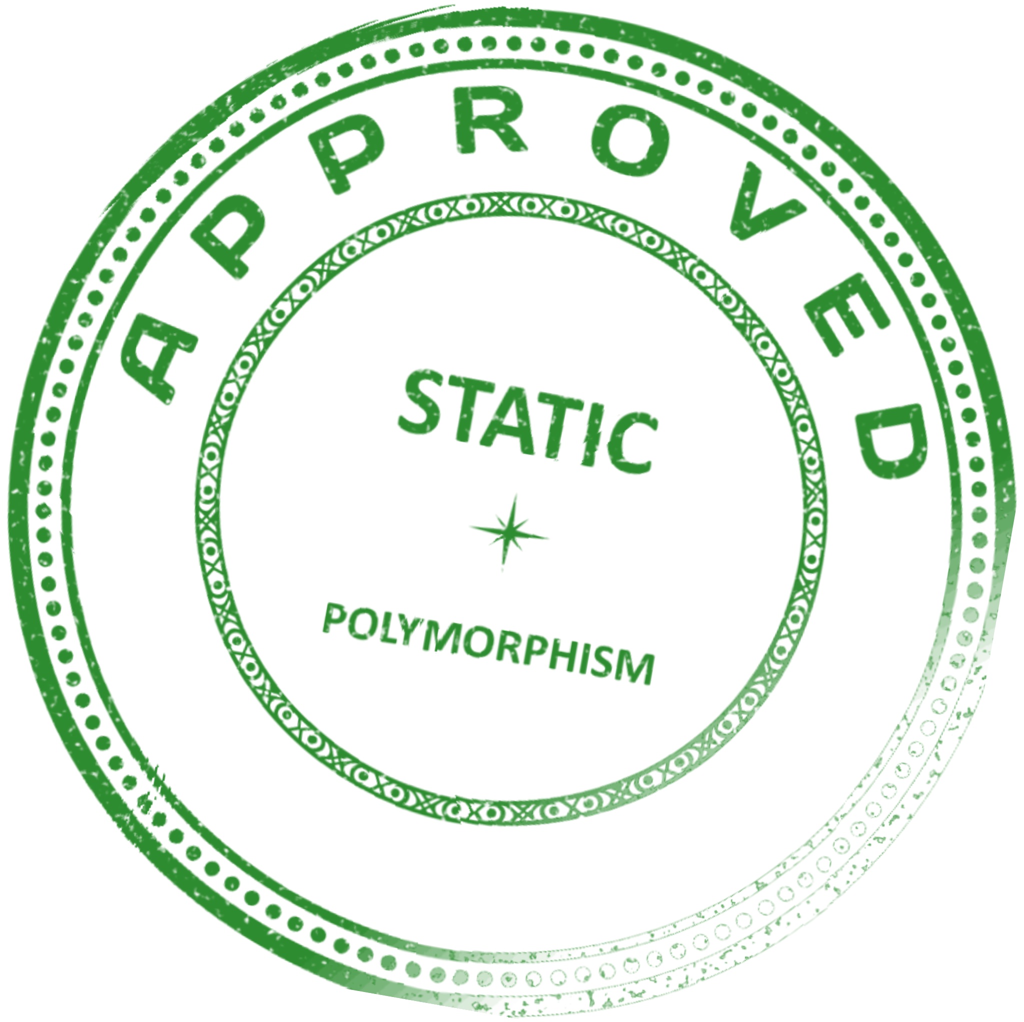
Function template:

```
struct point { int x; int y; int z; };

template<typename T>
std::string serialize(T object)
{
  // ...
}

auto p = point{1, 2, 3};
auto v = std::vector<point>{{4, 5, 6}, {7, 8, 9}};

auto result1 = serialize(1729);
auto result2 = serialize(p);
auto result3 = serialize(v);
```



Subtyping

Inheritance and virtual functions:

```cpp
struct animal {
  virtual void speak() = 0;
};

struct dog : animal {
  virtual void speak() override { /* ... */ }
}

struct cat : animal {
  virtual void speak() override { /* ... */ }
}

void say_something(animal *a)
{
  a->speak();
}
```

Subtyping + Static polymoprhism = ?

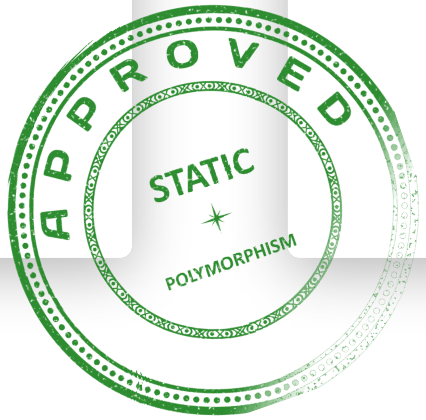
Subtyping + Static polymoprhism = CRTP

CRTP (Curiously recurring template pattern)

```cpp
template <typename T>
struct base
{
  void interface()
  {
    static_cast<T*>(this)->impl();
  }

  static void static_func()
  {
    T::static_sub_func();
  }
};
```

```cpp
struct derived : base<derived>
{
  void impl()
  {
    // ...
  }
  static void static_sub_func()
  {
    // ...
  }
};
```



Function templates and function overloading

Function overloading

```cpp
void print_me(int i) // #1
{
  // ...
}
void print_me(double d) // #2
{
  // ...
}
void print_me(std::string s) // #3
{
  // ...
}

print_me(17.0);         // calls #2
print_me(8);            // calls #1
print_me("2022"s);      // calls #3
```

Function overloading — function templates

```cpp
template<typename T>
void sort(T t)
{
  // ...
}

std::vector v = {3, 2, 1};
std::list l = {6, 5, 4};

sort(v);
sort(l);
```

Function overloading — function templates

```cpp
template<typename T>
void sort(std::vector<T> v) // random access
{
  // ...
}

template<typename T>
void sort(std::list<T> l) // bidirectional access
{
  // ...
}

std::vector v = {3, 2, 1};
std::list l = {6, 5, 4};

sort(v);
sort(l);
```

Function overloading — function templates

```cpp
template<typename C>
void sort_impl(C collection, slow_algo)
{
  std::cout << "slow\n";
}

template<typename C>
void sort_impl(C collection, fast_algo)
{
  std::cout << "fast\n";
}

template<typename C>
void sort(C collection)
{
  sort_impl(collection, which_algo_t<C>{});
}

std::vector v = {3, 2, 1};
std::list l = {6, 5, 4};

sort(v);
sort(l);
```

Function overloading — function templates

Tag dispatching

```cpp
template<typename C>
void sort_impl(C collection, slow_algo)
{
  std::cout << "slow\n";
}

template<typename C>
void sort_impl(C collection, fast_algo)
{
  std::cout << "fast\n";
}

template<typename C>
void sort(C collection)
{
  sort_impl(collection, which_algo_t<C>{});
}

std::vector v = {3, 2, 1};
std::list l = {6, 5, 4};

sort(v);
sort(l);
```

```cpp
struct fast_algo{};
struct slow_algo{};

template<typename C>
struct which_algo
{
  using type = slow_algo;
};

template<typename T>
using which_algo_t = typename which_algo<T>::type;

template<typename T>
struct which_algo<std::vector<T>>
{
  using type = fast_algo;
};
```

Function overloading — function templates

SFINAE: Substitution failure is not an error

```cpp
struct fast_algo{};
struct slow_algo{};

template<typename C>
struct which_algo
{
  using type = slow_algo;
};

template<typename T>
using which_algo_t = typename which_algo<T>::type;

template<typename T>
struct which_algo<std::vector<T>>
{
  using type = fast_algo;
};

template<typename T>
inline constexpr bool is_fast_v =
  std::is_same_v<which_algo_t<T>, fast_algo>;
```

Function overloading — function templates

SFINAE: Substitution failure is not an error

```cpp
struct fast_algo{};
struct slow_algo{};

template<typename C>
struct which_algo
{
  using type = slow_algo;
};

template<typename T>
using which_algo_t = typename which_algo<T>::type;

template<typename T>
struct which_algo<std::vector<T>>
{
  using type = fast_algo;
};

template<typename T>
inline constexpr bool is_fast_v =
  std::is_same_v<which_algo_t<T>, fast_algo>;
```

```cpp
template<typename C>
std::enable_if_t<is_fast_v<C>> sort(C collection)
{
    std::cout << "fast\n";
}

template<typename C>
std::enable_if_t<!is_fast_v<C>> sort(C collection)
{
    std::cout << "slow\n";
}

std::vector v = {3, 2, 1};
std::list l = {6, 5, 4};

sort(v); // fast
sort(l); // slow
```

Function overload resolution

- Tag dispatching
- SFINAE
- Expression SFINAE (detection idiom, ...)

Concepts to the rescue

Concepts are an extension to the templates feature provided by the C++ programming language. Concepts are named Boolean predicates on template parameters, evaluated at compile time. A concept may be associated with a template (class template, function template, member function of a class template, variable template, or alias template), in which case it serves as a constraint: it limits the set of arguments that are accepted as template parameters.

— Wikipedia, August 2022

Concepts are an extension to the templates feature provided by the C++ programming language. Concepts are named Boolean predicates on template parameters, evaluated at compile time. A concept may be associated with a template (class template, function template, member function of a class template, variable template, or alias template), in which case it serves as a constraint: it limits the set of arguments that are accepted as template parameters.

— Wikipedia, August 2022

Concepts

```cpp
template<typename T>
concept SupportsFastAlgo = is_fast_v<T>;

template<typename T>
concept SupportsSlowAlgo = !is_fast_v<T>;

template<typename C> requires SupportsFastAlgo<C>
void sort(C collection)
{
    std::cout << "fast\n";
}

template<typename C> requires SupportsSlowAlgo<C>
void sort(C collection)
{
    std::cout << "slow\n";
}

std::vector v = {3, 2, 1};
std::list l = {6, 5, 4};

sort(v);
sort(l);
```

Concepts

```cpp
template<typename T>
concept SupportsFastAlgo = is_fast_v<T>;

template<typename T>
concept SupportsSlowAlgo = !is_fast_v<T>;

template<SupportsFastAlgo C>
void sort(C collection)
{
    std::cout << "fast\n";
}

template<SupportsSlowAlgo C>
void sort(C collection)
{
    std::cout << "slow\n";
}

std::vector v = {3, 2, 1};
std::list l = {6, 5, 4};

sort(v);
sort(l);
```

Concepts

```cpp
template<typename T>
concept SupportsFastAlgo = is_fast_v<T>;

template<typename T>
concept SupportsSlowAlgo = !is_fast_v<T>;

void sort(SupportsFastAlgo auto collection)
{
    std::cout << "fast\n";
}

void sort(SupportsSlowAlgo auto collection)
{
    std::cout << "slow\n";
}

std::vector v = {3, 2, 1};
std::list l = {6, 5, 4};

sort(v);
sort(l);
```

Let us see more interesting examples…

The end game: What are the advatages?

- Requirements for templates are part of the interface
- The overloading of functions or specialisation of class templates can be based on concepts
- We get an improved error message because the compiler compares the requirements of the template parameter with the actual template arguments
- You can use predefined concepts or define your own
- The usage of *auto* and concepts is unified. Instead of *auto*, you can use a concept.
- If a function declaration uses a concept, it automatically becomes a function template. Writing function templates is, therefore, as easy as writing a function.

Thank you