C++20 `concept` – The Base Classes of Generic Programming? Or Not?

Ankur M. Satle

ankursatle@gmail.com

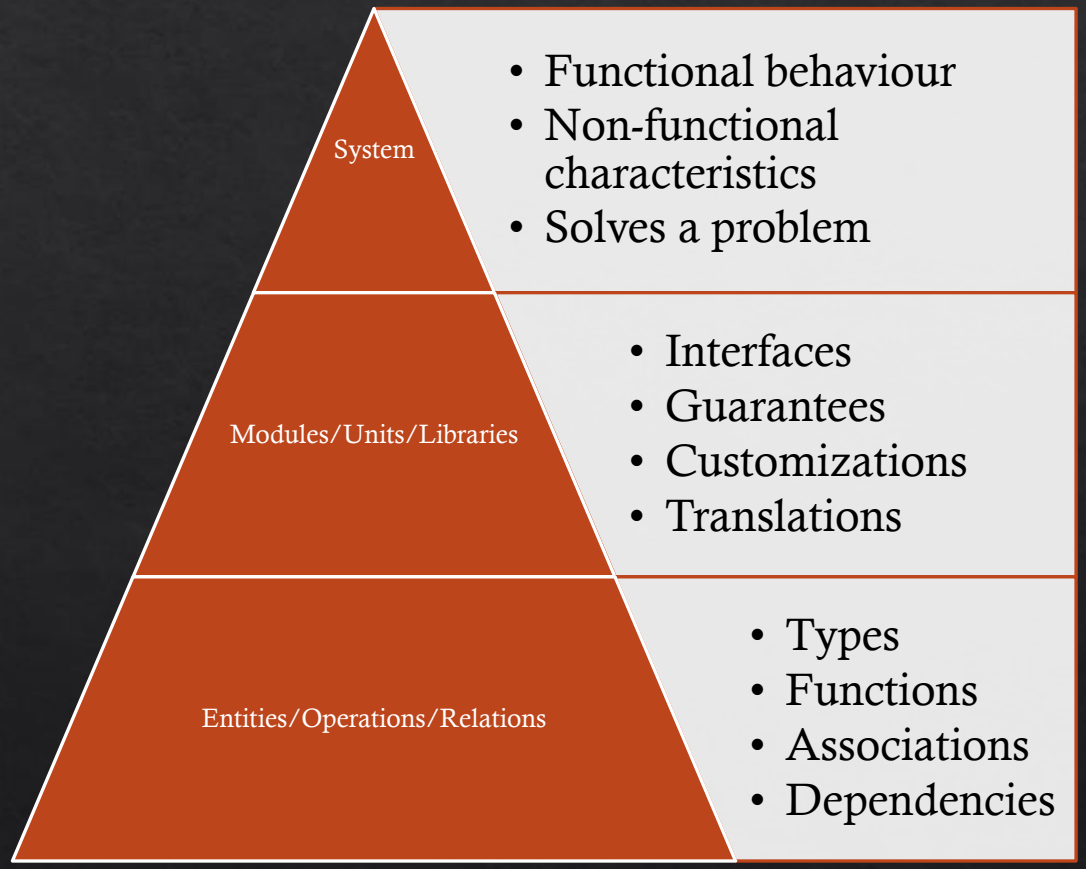
$whoami

- Architect at EXFO
- Love C++
- Performance focussed
- Working on Cloud Native products
- Neophilia

- https://ankursatle.wordpress.com
- ankursatle@gmail.com

Scalable systems

- Strategies to deal with the scale:
  - Divide and conquer
  - Abstractions
    - Types
    - Operations
- We also strongly desire
  - Reusability
  - Cohesion
  - Dependency Inversion



OOP World

- At the implementation level
  - Entity
  - Relationships
- Characterised by
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

Base Classes of the OOP World

- Poster child of OOP

```
class Shape
{
   virtual void draw() const = 0;
};
class Circle : public Shape
{
   virtual void draw() const {
      /* Implementation */
   }
};
class Square : public Shape {};
```

- This allows us to write generic code

```
void show(const Shape& shape) {
   shape.draw();
}
```

- Circle is-a Shape; Square is-a Shape
- Write once, use many times, with different types – polymorphism
- Common way to work with any sub-class of Shape

Concept of the Generic Programming World

- Concepts look so like Base classes!

```
template<typename T>
concept Shape =
    requires(const T& shape) {
        shape.draw();
};
class Circle : public Shape
{
    virtual void draw() const {
        /* Implementation */
    }
};
class Square {};
```

- We still want this generic code

```
void show(const Shape auto& shape) {
    shape.draw();
}
```

- Generic!
- Wait, exactly the usage code with Base class!?
- They look so similar! But are they!?

Concepts – what are they?

- The concept

```
template<typename T>
concept Shape =
   requires(const T& shape) {
      shape.draw();
};
```

- When used like this

```
void show(const Shape auto& shape) {
   shape.draw();
}
```

- Can be seen as a short-hand for:

```
template<typename T>
void show(const T& shape) {
   shape.draw();
}
```

- Concepts are nothing but constrained templates

`template` vs `concept`

- A function template

```
template<typename T>
void show(const T& shape) {
    shape.draw();
}
```

- What types can we pass to this function?
- Let's try one out

```
Employee e;
show(e);
```

- Compilation output:

```
<source>: In instantiation of 'void
show(const T&) [with T = Employee]':
<source>:19:9:   required from here
<source>:10:11: error: 'const class
Employee' has no member named 'draw'
   10 |     shape.draw();
      |     ~~~~~~^~~~
Execution build compiler returned: 1
```

- Templates are the **`void*`** of C++

template vs concept

- With Concepts?

```
template<typename T>
concept Shape = requires(const T& shape) {
    shape.draw();
};
void show(const Shape auto& shape) {
    shape.draw();
}
```

- When used with an incorrect type:

```
Employee e;
show(e);
```

- Concepts state expectations from a type
- Mismatched type usage results in a short, easy error
- Error shown in the interface rather than within the guts of the implementation

- Compilation gives:

```
<source>:21:5: error: no matching function for
call to 'show'


<source>:11:6: note: candidate template
ignored: constraints not satisfied [with
shape:auto = Employee]

<source>:11:17: note: because 'Employee' does
not satisfy 'Shape'


<source>:9:50: note: because 'shape.draw()'
would be invalid: no member named 'draw' in
'Employee'

concept Shape = requires(const T& shape) {
shape.draw(); };
```

Constraints

- When you require a type to be constructible from another

```
struct json{ /* … */ };

template<typename T>

concept http_resp = requires(json body) {

      T::T(body);

};

struct resp {

    resp(const json&);

};

http_resp auto hr = resp(json{});
```

- On compilation

```
<source>:38:5: error: deduced type 'resp' does not satisfy 'http_resp'

    http_resp auto hr = resp(json{});

    ^~~~~~~~~

<source>:12:8: note: because 'T::T(body)' would be invalid: no member named 'T' in 'resp'

    T::T(body);

       ^

1 error generated.
```

- Strangely, this looks for a function named T!

Constraints

- The right way is to use another concept

```
#include <concepts>

struct json{ /* … */ };

template<typename T>

concept http_resp =
std::constructible_from<T, json>;

struct resp {

    resp(const json&);

};

http_resp auto hr = resp(json{});
```

- On compilation

```
Al izz well... ☺
```

- A concept can use another concept to model an aspect or part
- Concepts are composabile!

Constraints

- In fact, multiple constraints can be added using and/or

```
#include <concepts>

struct json{ /* … */ };

template<typename T>

concept http_resp =
std::constructible_from<T, json> &&
std::is_move_constructible_v<T>;

struct resp {

    resp(const json&);

};

http_resp auto hr = resp(json{});
```

- A concept can use a type_trait to constrain an aspect or part
- Concepts can use type_traits
- Multiple constraints can be used in conjunction or disjunction!

Ensuring a type models a concept

- No mechanism in the type definition. Following ensures.

```
static_assert(Sender<Socket_sender>, "Socket_sender does not model concept Sender correctly");
```

- If the type does not fully model the concept, the compiler outputs:

```
error: static assertion failed: Socket_sender does not model concept Sender correctly
   36 | static_assert(Sender<Socket_sender>, "Socket_sender does not model concept Sender correctly");
      |               ^~~~~~~~~~~~~~~~~~~~
<source>:36:15: note: constraints not satisfied
<source>:13:9:   required by the constraints of 'template<class T> concept Sender'
<source>:13:18:   in requirements with 'const T& t', 'std::span<std::byte> m', 'alternatives& alt' [with T = Socket_sender]
<source>:22:12: note: the required expression 'T::Send()' is invalid
   22 |     T::Send();
      |     ~~~~~~~^~
```

- Compiler does a better job!

Multiple Interfaces

- Multiple inheritance possible in OOP
- A type satisfies many Base classes

```
class HttpServer : public
Sender, public Receiver
{};
HttpServer hs;
Sender& s = hs;
Receiver& r = hs;
```

- Possible to model many concepts
- static_assert to enforce each otherwise non-conformance will be raised on use

```
static_assert(Sender<HttpServer>);
static_assert(Receiver<HttpServer>);
```

- Or create a super concept

```
template<typename T>
concept SenderReceiver =
    Sender<T> && Receiver<T>;
static_assert(
    SenderReceiver<HttpServer>);
```

Supporting a new type

- OOP
  - Must be derived from the same base
  - May not have control over new type
- Concepts
  - New types fit in well just by themselves
  - No need to be intrusive using inheritance
  - If the type models the concepts without the structural overhead, it will work

Containers of Base Type

- Poster child of OOP

```
class Shape
{
   virtual void draw() const = 0;
};
class Circle
{
   virtual void draw() const {
      /* Implementation */
   }
};
```

- This allows us to write generic code

```
const std::vector<Shape*> shapes;


void show(const std::vector<Shape*>
shapes) {
   for (const auto& shape : shapes) {
      shape->draw();
   }
}
```

- Smooth and polymorphic

Containers of Generic Type

- Poster child of OOP

```
concept Shape = requires{
draw(); };

class Circle
{
   void draw() const {};
};
```

- Is this possible?

```
const vector<Shape auto*> shapes;


void show(const std::vector<Shape
auto*> shapes) {
   for (const auto& shape : shapes) {
      shape->draw();
   }
}
```

- Can be of only one type

Base Class may provide a default impl

- Base classes can provide a default impl that derived types may not themselves implement

```
class Msg {
   virtual string to_string()
   { /*Default Impl*/ }
};
class HttpMsg : public Msg
{ /* No to_string() */ };

HttpMsg req;
send(req.to_string());
```

- Concepts are just a set of constraints
- Concepts cannot provide a default implementation to the types modelling them

```
template<typename T>
concept serializable = requires(T t){
{t.to_string()} -> same_as<string> };
class SmsMsg { /* No to_string() */ };
static_assert(serializable<SmsMsg>);
//error: no 'to_string'
static_assert(serializable<HttpMsg>);
```

- A type models a concept even if it's impl comes from Base class

Overload on cv-qualifiers

- It is possible to overload functions on cv-qualifiers

```
class Msg {
   virtual void process();
   virtual void process() const;
};
class HttpMsg : public Msg
{};

const HttpMsg msg;
msg.to_string();  //const called
```

- Concepts too allow for cv-sensitive constraints

```
template<typename T>
concept http_req = requires
   (const T t_const, T t) {
      t_const.process();
      t.process();
};
struct req { resp process(); };
static_assert(http_req<req>);
```

- cv-qualifier sensitive constraints possible

noexcept specifications

- Concepts may specify noexcept expectations

```
template<typename T>
concept http_req = requires(const T req) {
      { req.process() } noexcept;
      { req.process() } const noexcept;
};
struct req { resp process() const noexcept; };
static_assert(http_req<req>);
```

- cv-qualifier sensitive constraints possible

Overload on different passed types

- Given

```
class Sender {
   void send(const std::string&);
   void send(std::string&&);
};
```

- Different functions will be called based on the type of data passed

```
template<typename T>
concept Sender = requires(T
sender, const std::string& s_cref,
std::string&& s_rvalue,
std::string svalue) {
    sender.send(s_cref);
    sender.send(s_rvalue);
    sender.send(svalue); //Generic
};
```

- Explicit checks on the specific types possible

Overload accepting variant

- In fact, even if the type accepted a variant

```
class Sender {
   void send(std::variant<
       std::string, int, float>);
};
```

- Different functions will be called based on the type of data passed

```
template<typename T>
concept Sender = requires(T
sender, std::string svalue) {
   sender.send(svalue);
};
```

- Concepts allow you to stay at semantic level and allow for syntax to vary
- Efficiency specific constructs do not come in the way

Class static function

- A Base class will not be able to specify whether a class static function is expected in its Derived types
- Class static functions cannot be virtual

```
Derived::get_class_stats();
```

- With concepts

```
template<typename T>
concept Sender = requires {
    T::getInstance();
};
```

- It is possible to check for class static functions also which Base classes cannot demand from it's derived types

Default argument

- Concept matches with binding functions with default arguments

```
template<typename T>
concept Sender = requires(T sender) {
    sender::send();
};

class SmsSender {
   void send(const std::string& msg = welcome_msg);
};
```

No structural elements come from concept

- Base class can provide structural commonality

```
struct Person {
    std::string name;
};
struct Employee : public Person {
};
std::cout << employee.name;
```

- Concept do not affect the structure of the type
- Concepts can specify member variable expectations though

```
template<typename T>
concept Person = requires {
    T::name;
};
```

Constraint on a member type

- Concepts can very well expect a member type

```
template<typename T>
concept Container = requires {
   typename T::value_type;
};

static_assert(Container<std::vector<int>>);
```

Constraining the return-type of a function

- Concepts can specify the return type of functions

```
template<typename T>
concept Message = requires (T t) {
    { T::to_string() } -> std::same_as<std::string>;
    { T::to_json() } -> std::convertible_to<json>;
    { T::to_binary() } -> std::derived_from<byte_buffer>;
};
```

Base* can point to a different Derived*

- Pointer to a Base can point to a Derived class
- It can be changed to point to another Derived type
- This works well – rebinding every time

```
Base* bptr = d1_obj1;

bptr = d1_obj2;

bptr = d2_obj1;
```

- With concepts, the data-types become concrete on instantiation
- A concept pointer can point to another object of the same type
- But not to an object of different type

```
Msg auto* msgptr = sms_msg1;

msgptr = sms_msg2;

msgptr = txt_msg1;

//error: no conversion from
TxtMsg to SmsMsg
```

Private, protected and friends

- Private, protected & public control access to parts of a class
- This visibility also impacts concept use
- A type may match different concept depending upon the visibility
- The same object may match different concepts depending on the visibility
- Beware of this possible trap

Checking for Derived/Modelling data-type

- Checking for Derived types

```
class Base { virtual ~Base();};
class D1: public Base {};
class D2: public Base {};

Base* bp = get_some_obj();
if (dynamic_cast<D1*>(bp))
{ /* It is a D1 */ }
```

- Checking if an object is a certain type

```
template<typename T>
concept Container = requires{};

Container auto& c = get_a_cont();
if constexpr
(std::same_as<decltype(c), Type1>)
{ /* c is Type1 */ }
```

- Checking for conversion, is similar
- Better that it happens at compile time – compiled code will always be optimal!

Overloading with different concepts

- Vary concepts to have different flavours

```
template<typename T>
concept Sender = requires(T sndr)
{ sndr.send(); };

template<typename T>
concept EncryptedSender =
Sender<T> && requires(T sndr)
{ sndr.encrypt(); };

sendMsg(Sender auto& s, Msg m);
sendMsg(EncryptedSender auto& s,
Msg m);
```

- Overload resolution based on type modelling the specific concept

```
ConcreteSender sender;
Msg m;
sendMsg(sender, m);
```

- Short answer: the more specific one
- But how? Subsumption rules apply.

Relationships

- Concepts can help specify expected constructs out of the type too

```
struct json{ /* … */ };

template<typename T>
concept http_resp =
std::constructible_from<T, json>;


template<typename T, typename R>
concept req_processing =
std::constructible_from<T, json> &&
requires(T req) {
    { apply(req) } -> http_resp;
};
```

```
struct resp { resp(const json&); };
struct req { req(const json&); }


static_assert(req_processing<req,
resp>);
```

- concept can help formulate the program structure
- Very powerful
- Not just restricted to syntactic requirements on a single type/Base class

Surprise!

- What have we got?

```
template<typename T>
concept Sender = requires(const T& t, std::span<std::byte> m) {
    t.send(m);
};
class SmsSender
{
   void send(std::span<std::byte>) const {
       /* Implementation */
   }
};
```

- Generic code

```
void inform_user(const Sender auto& sender) {
   sender.send();
}
```

- This compiles to our surprise!

- The above can be used as:

```
SmsSender sms_sender;
inform_user(sms_sender);
```

- Error only on use of `inform_user`
- Concepts are nothing but constrained templates

Surprise!

- Poster child of OOP

```
template<typename T>
concept Sender = requires(const T& t,
std::span<std::byte> m) {
    t.send(m);
};
class SmsSender
{
   void send(std::span<std::byte>)
const {
       /* Implementation */
   }
};
```

- Generic code

```
enum class Event {
   bill_generated, payment_reminder,
   credit_limit_exceeded
};
void inform_user(const Sender auto& sender,
Event event) {
   auto message = create_sms(event);
   sender.send(message);
}
```

- Which can be used as:

```
SmsSender sms_sender;
inform_user(sms_sender, payment_reminder);
```

- Here, if the type of message is not

Guideline – name finer concepts

- Possible to define a constraint inline
- While this is convenient, it should be done for trivial cases only
- Subsumption cannot happen with this
- Name them but not every item!

Supporting existing non-conforming types

- Implement conversion
- Use Adapter
- Create a Base class

- Implement Type Conversion
- Use Accessor
- Create a concept

- Provide a customization point

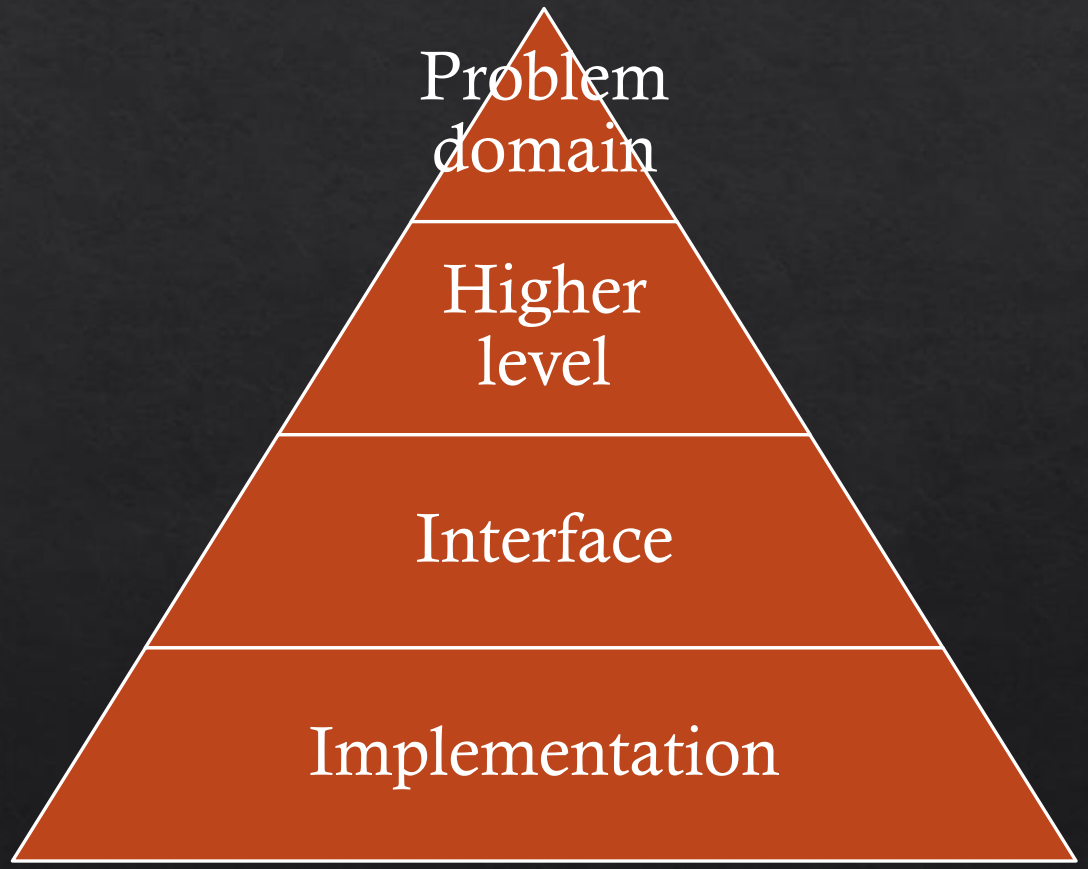
How to organize concepts

- Keep domain concepts and technical concepts separate
- Copyable, Non-Copyable, Serializable, Loggable, Auditable may be core implementation-level constraints
  - not core to the domain/business logic
  - these should not by default be a part of the domain level concept
- Combine & build upon where the ideas are central to the higher-level
- Beware of having silent/hidden expectations not specified in the concept
- Be cognizant of breaking agreement with the user of the library

Namespaces for organization & layering

- We manage different layers of the applications in different namespaces
- Same can be used for concepts too

- Concepts enable structuring higher-level structure of program
- Concepts provide semantic specifications to constructs rather than specify low-level syntax

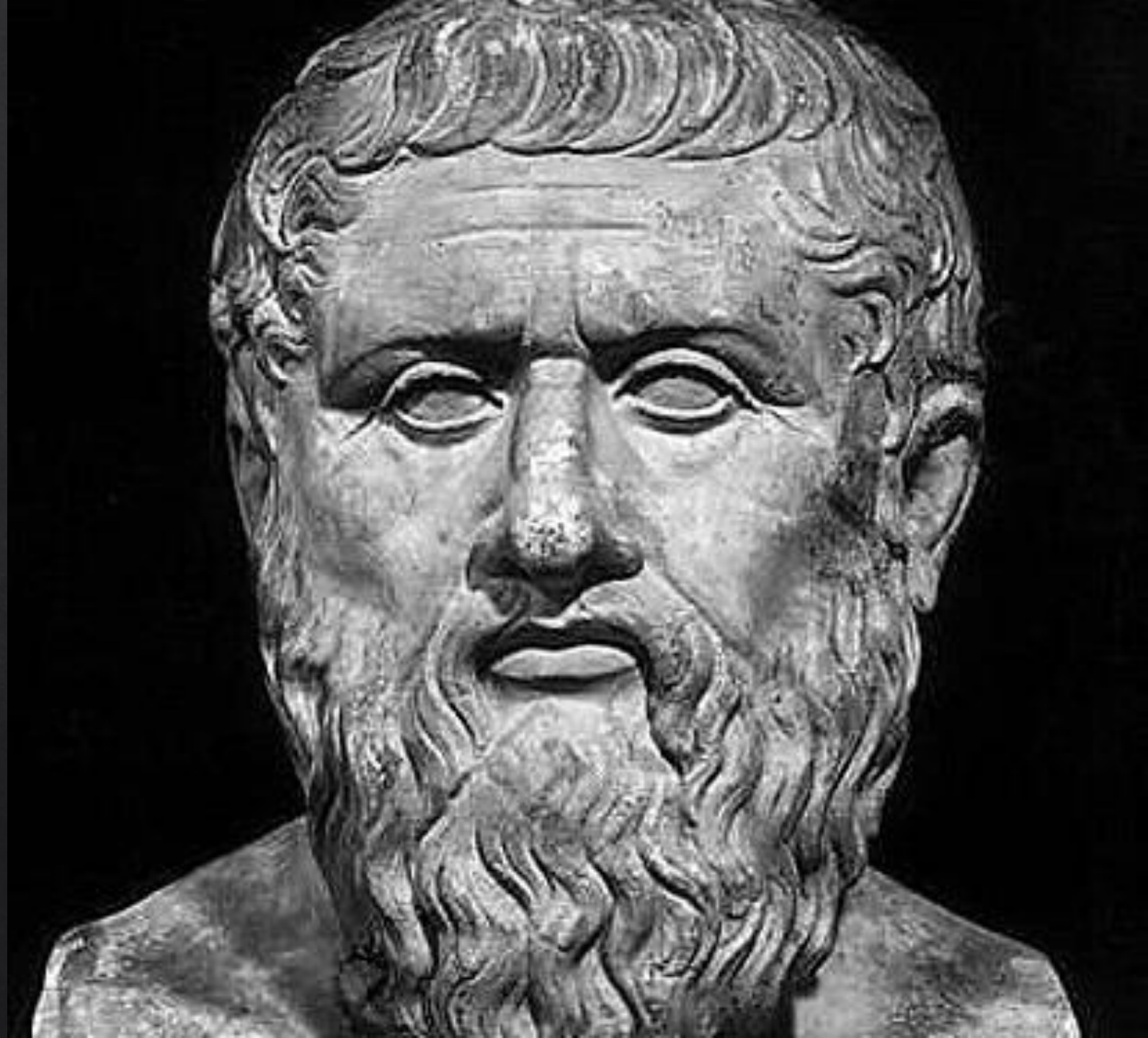


“

Man is a being in search of meaning.

”

Plato



Other excellent resources on concepts

- C++Now 2021 talks by Jeff Garland “Using Concepts: C++ Design in a Concept World”
- C++Now 2021 talk by Andrzej Krzemienski on “So You Think You Know How To Work With Concepts?”
- Many other syntax-focused talks
- The C++ Standard

I do not have time to make all the mistakes myself

Please share your learning…

ankursatle@gmail.com

समाप्त

धन्यवाद