TDDD38 - Possible features in coming standards

Eric Elfving

Department of Computer and information science

LINKÖPING UNIVERSITY

LINKÖPING
UNIVERSITY

Assume that we have the following class to represent a point:

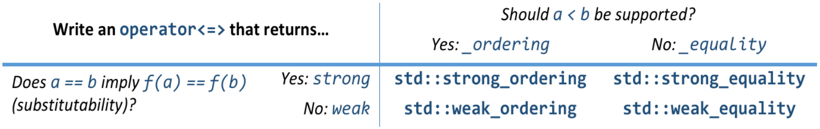
```
class Point {
    int x;
    int y;
};
```

We now want to add comparison between `Point` objects. This usually is done by implementing `operator<` and `operator==` and then using these to implement the others...

```
bool operator==(Point p1, Point p2) {
    return p1.x == p2.x && p1.y == p2.y;
}
bool operator<(Point p1, Point p2) {
    Point origin {0,0};
    return distance(p1, origin) < distance(p2, origin);
}
bool operator>(Point p1, Point p2){
    return p2 < p1;
}
bool operator<=(Point p1, Point p2){
    return !(p1 > p2);
}
bool operator>=(Point p1, Point p2){
    return !(p1 < p2);
}
bool operator!=(Point p1, Point p2){
    return !(p1 == p2);
}
```

- In C++20, we'll get the "spaceship operator" (threeway comparison).
- Basically, we create one operator, `operator<=>`, and the rest can be generated.
- The return type should be convertable to int:
  - `a<=>b` $< 0 \iff a < b$
  - `a<=>b` $= 0 \iff a = b$
  - `a<=>b` $> 0 \iff a > b$

Several types are provided in `<compare>` to be used instead for int



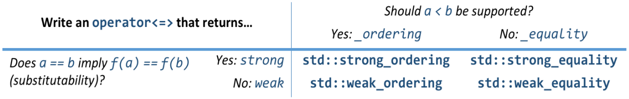
Write an `operator<=>` that returns...		*Should a < b be supported?*	
		Yes: _ordering	*No: _equality*
Does a == b imply f(a) == f(b) (substitutability)?	*Yes: strong*	`std::strong_ordering`	`std::strong_equality`
	No: weak	`std::weak_ordering`	`std::weak_equality`

Each have static members to get good names (and more expressive code!):

```
struct strong_ordering {
    static constexpr int less    = /* a negative value */;
    static constexpr int equal   = 0;
    static constexpr int greater = /* a positive value */;
    ...
};
```

- Let’s implement `operator<=>`!

- Let’s implement `operator<=>` !
- What return type should we support?



Write an `operator<=>` that returns...		*Should a < b be supported?*	
		Yes: `_ordering`	*No: `_equality`*
Does a == b imply f(a) == f(b) (substitutability)?	*Yes: `strong`*	**`std::strong_ordering`**	**`std::strong_equality`**
	No: `weak`	**`std::weak_ordering`**	**`std::weak_equality`**

- Let’s implement `operator<=>` !
- What return type should we support?
- => `std::strong_ordering`

- Let’s implement `operator<=>`!

```
std::strong_ordering operator<=>(Point p1, Point p2)
{
    if ( p1.x == p2.x && p1.y == p2.y )
    {
        return strong_ordering::equal;
    }
    Point origin {0,0};
    if (distance(p1, origin) < distance(p2, origin))
    {
        return strong_ordering::less;
    }
    return strong_ordering::greater;
}
```

We could also implement it by calling `operator<=>`:

```
std::strong_ordering operator<=>(Point p1, Point p2)
{
    if ( p1.x == p2.x && p1.y == p2.y )
    {
        return strong_ordering::equal;
    }
    Point origin {0,0};
    return distance(p1, origin) <=>
             distance(p2, origin);
}
```

- But, what if `operator<=>` isn't defined for the type returned from `distance`?

- But, what if `operator<=>` isn't defined for the type returned from `distance`?
- => use `std::compare_3way`.

- But, what if `operator<=>` isn't defined for the type returned from `distance`?
- => use `std::compare_3way`.

```
std::strong_ordering operator<=>(Point p1, Point p2)
{
    if ( p1.x == p2.x && p1.y == p2.y )
    {
        return strong_ordering::equal;
    }
    Point origin {0,0};
    return std::compare_3way(distance(p1, origin),
                             distance(p2, origin));
}
```

The compiler can generate the operator iff we want to compare subobjects (bases and members) lexicographically (i.e. use `<=>` between each pair of subobjects until a result is found).

```
struct Time {
    int hour, minute, second;
    auto operator<=>(Time const &) const = default;
};
```

Name lookup

The old rules state that the binary operator `@` (when used in an expression like `a @ b`) should be found as `a.operator@(b)` or else `operator@(a,b)`. The new rules does name lookup for the following expressions as well `a <=> b` and `b <=> a`. They should be preferred in this order if there are several matches.

- If `a <=> b` is the best match, transform the expression to `a<=>b @ 0`
- If `b <=> a` is the best match, transform the expression to `0 @ b<=>a`

Larger example

Comparison for `std::optional` by Barry Revzin

```
template <typename T>
class optional {
public:
    template <typename U>
    constexpr auto
    operator<=>(optional<U> const& rhs) const
      -> decltype(compare_3way(**this, *rhs))
    {
        if (has_value() && rhs.has_value()) {
            return compare_3way(**this, *rhs);
        } else {
            return has_value() <=> rhs.has_value();
        }
    }
```

Larger example

Comparison for `std::optional` by Barry Revzin

```
    template <typename U>
    constexpr auto
    operator<=>(U const& rhs) const
      -> decltype(compare_3way(**this, rhs))
    {
        if (has_value()) {
            return compare_3way(**this, rhs);
        } else {
            return strong_ordering::less;
        }
    }
```

Larger example

Comparison for `std::optional` by Barry Revzin

```
    constexpr strong_ordering
    operator<=>(nullopt_t ) const
    {
        return has_value() ? (
             strong_ordering::greater
             : strong_ordering::equal);
    }
};
```

References

- P0515 - Consistent comparison.
- Cppreference overview.

LINKÖPING UNIVERSITY

Contracts is a way of specifying preconditions, postconditions and assertions required for a function as a part of it’s signature

- Preconditions (`expects`) is a predicate that is supposed to hold when entering the function.
- Postconditions (`ensures`) is a predicate that holds upon exiting the function.
- Assertions (`assert`) are checked in a function during computation.

Preconditions and postconditions are part of the external view of the function and are placed before the function body while assertions are placed inside the function.

Short example

```
void push(queue & q)
    [[ expects: !q.full() ]]
    [[ ensures: !q.empty() ]]
{
    // ...
    [[ assert: q.is_ok() ]];
}
```

We require that the queue isn’t full at the beginning and assures the caller that it’s not empty afterwards. Somewhere inside we check for validity.

- This is intended as a runtime check
- But of course static analysers and compilers could check it as well
- Three degrees (or levels) of checking is proposed (`expects` used as example):
    - `[[expects default: ...]]` - unless turned off, the contract will be checked during runtime

- This is intended as a runtime check
- But of course static analysers and compilers could check it as well
- Three degrees (or levels) of checking is proposed (`expects` used as example):
  - `[[expects default: ...]]` - unless turned off, the contract will be checked during runtime
  - `[[expects: ...]]` - implicit shorthand for default

- This is intended as a runtime check
- But of course static analysers and compilers could check it as well
- Three degrees (or levels) of checking is proposed (`expects` used as example):
  - `[[expects default: ...]]` - unless turned off, the contract will be checked during runtime
  - `[[expects: ...]]` - implicit shorthand for default
  - `[[expects axiom: ...]]` - not checked at runtime, purely as comment or for static analyzers. No runtime cost.

- This is intended as a runtime check
- But of course static analysers and compilers could check it as well
- Three degrees (or levels) of checking is proposed (`expects` used as example):
    - `[[expects default: ...]]` - unless turned off, the contract will be checked during runtime
    - `[[expects: ...]]` - implicit shorthand for default
    - `[[expects axiom: ...]]` - not checked at runtime, purely as comment or for static analyzers. No runtime cost.
    - `[[expects audit: ...]]` - a more comprehensive (and expensive) check than default. Must be specifically turned on.

- Default behavior when breaking a runtime contract check is to terminate.
- There will also be a (implementation specified) way of providing your own handler.

References

- P0380 - A Contract Design.

LINKÖPING UNIVERSITY

There are two main ways of handling errors in C++; exceptions and error codes. We have had exceptions for over 25 years, but a recent developer survey[1] shows that 52% of respondents work in code that banns exceptions in total or in part!

[1] isocpp.org developer survey 2018

Exceptions

- Are the only way of signalling errors in constructors and in operators
- Are used heavily in the STL
- Violates both the zero overhead principle and determinism!
    - Requires RTTI (both `typeid` and `dynamic_cast`)
    - Requires extra storage - both in the binary and during runtime
    - Handling several exceptions is tricky and gives unpredictable compile-time space and time cost

Exceptions

”I can’t recommend exceptions for hard real time; doing so is a research problem, which I expect to be solved within the decade”
Stroustrup 2004

Error codes

The C solution is returning an error code (often a simple type such as int or enum):

```
error_code fun(actual_return_type & val)
{
    // either modify val and return "no error"
    // or return specific error code
}
```

```
return_type fun(error_code & err) // or pointer
{
    // either set err to "no error" and return
    // a valid value or set error code and return
    // a dummy value
}
```

There are several problems with error codes:

- The user has to check the error code (at all calls)
- Will always require space for both error and actual value
- Doesn’t work in constructors or operators - leads to a language variant where the user has to check if the current object actually is constructed (or calling factory methods instead of constructors)

All lead to modified control flow which hide the actual business logic behind error handling.

The committee has added some support already when working with error codes:

- `[[nodiscard]]`
  The compiler should warn when the return value from a function marked nodiscard is ignored:

```
[[nodiscard]] error_code fun();

// ...
    fun(); // warning
    if ( fun() == error::ok ) //ok
        ...
```

The committee has added some support already when working with error codes:

- `[[nodiscard]]`
  The compiler should warn when the return value from a function marked nodiscard is ignored:
- `std::error_code` - error message from the system, stores a category and a value.

The committee has added some support already when working with error codes:

- `[[nodiscard]]`
  The compiler should warn when the return value from a function marked nodiscard is ignored:
- `std::error_code` - error message from the system, stores a category and a value.
- `std::optional`? - not made for error handling, but could be used for it

expected

Proposal P0323 wants to add a new type `std::expected` that can be used for error handling.

```
std::expected<ReturnType, ErrorType> fun()
{
    if ( /* error */ )
        return std::unexpected{ErrorType::specific_error};
    return my_actual_value;
}
int main()
{
    auto e = fun();
    if ( !e )
    {
        if ( e.error().value() == ErrorType::specific_error )
            // handle this error
    }
    else
        // do something with *e
}
```

Throwing values!

Herb Sutter has proposed a new technique for throwing objects by values with zero (or in some cases negative) overhead!

- Mark the function with the new keyword `throws`
- The compiler will internally modify the return value to be a union containing either the actual return value or a value of type `std::error` and some information on whether we throw or return
- Code as usual in your function (return or throw)
- Catch `std::error` by value

example

```
string f() throws {
    if ( /* some error */ )
        throw arithmetic_error::some_error;
    return "hello"s; // possible dynamic error is transformed
}
string fun() // no "throws"
{
    return f(); // error converted to normal exception!
}
int main() {
    try {
        auto s = f();
        cout << s;
    }
    catch (error err) {
        if ( err == arithmetic_error::some_error )
          // handle this error
    }
}
```

Reflection on error handling

The combination of contracts and statically deterministic error codes would transform a lot of code for the better!

References

- P0323 - `std::expected`
- P0709 - Zero-overhead deterministic exceptions: Throwing values

LINKÖPING UNIVERSITY

Concepts WILL be in C++20!

- Basically a simple way of adding constraints on template parameters without having to use SFINAE!
- We add a `requires` clause to our template declaration and the compiler will check that the type provided fulfills the concept.
- The standard (with the ranges proposal) will add lots of good concepts to the STL, but we can create our own as well.

example

```
// Define a concept to check that given a value x of type T,
// ++x and x++ should be supported and both return an object of type T.
template <typename T>
concept Incrementable = requires ( T x ) {
    { x++ } -> T;
    { ++x } -> T;
};

// use concept
template <typename T>
void increment(T & val)
  requires Incrementable<T>
{
  ++x;
}
```

It’s also possible to use a static boolean expression as a concept:

```
template <typename T>
void fun(T val)
  requires is_integral_v<T>
{ ... }
```

Terse syntax

There are discussions on other syntax as well:

- Two from the original proposal

```
template <Incrementable T>
void increment(T & val);
```

```
void increment(Incrementable & val);
```

- In-place syntax [P0745]

```
void increment(Incrementable{} & val);
```

Concepts vs. tag dispatching

Listing 1: Tag dispatching

```
template <typename ForwardIt>
void advance(ForwardIt & it, size_t n,
             std::forward_iterator_tag) {
  while (n--) ++it;
}

template <typename RandomIt>
void advance(RandomIt & it, size_t n,
             std::random_iterator_tag){
  it += n;
}

template <typename It>
void advance(It & it, size_t n)
{
    using it_tag =
      typename iterator_traits<It>
                ::iterator_category;
    advance(it,n, it_tag{});
}
```

Concepts vs. tag dispatching

Listing 3: Tag dispatching

```cpp
template <typename ForwardIt>
void advance(ForwardIt & it, size_t n,
             std::forward_iterator_tag) {
  while (n--) ++it;
}

template <typename RandomIt>
void advance(RandomIt & it, size_t n,
             std::random_iterator_tag){
  it += n;
}

template <typename It>
void advance(It & it, size_t n)
{
    using it_tag =
      typename iterator_traits<It>
               ::iterator_category;
    advance(it,n, it_tag{});
}
```

Listing 4: concepts

```cpp
template <typename It>
void advance(It & it, size_t n)
  requires RandomAccessIterator<It>
{
    it += n;
}

template <typename It>
void advance(It & it, size_t n)
  requires ForwardIterator<It>
{
    while (n--) ++it;
}
```

References

- Rationale by Stroustrup P0557 - Concepts: The future of Generic Programming (or how to design good concepts and use them well)
- Standards text P0734

LINKÖPING UNIVERSITY

The Ranges TS has two main parts and will hopefully be merged into C++20:

- A huge set of predefined concepts
- Additions to the algorithms so that they can accept a range instead of a pair of iterators (the old syntax will of course not be removed). A range could be a container, but also something more...

Range adaptors

The range adaptor proposal adds support for pipelining transformations on a range.

```
auto square = [](int x){return x*x;};
int total =
   accumulate(view::iota(1) |               // generate a range starting at 1
                view::transform(square) |   // apply square to each
                view::take(10),             // stop after 10 elements
              0);
```

References

- Ranges TS N4685
- P1037 - Deep Integration of the Ranges TS
- P0789 - Range Adaptors and Utilities
- range-v3 implementation at github (with extra references)

LINKÖPING UNIVERSITY

The current model with preprocessor includes is simple to implement, but hard to work with. Every `#include` requires full parsing of the included header - even if we already used it before. We have to make sure not to have circular includes. It is also a very blunt way of specifying APIs!

In it's core, the module TS is rather simple:

- A file can declare itself as a module with the `module MODULE_NAME;` syntax.

In it’s core, the module TS is rather simple:

- A file can declare itself as a module with the `module MODULE_NAME;` syntax.
- A module explicitly exports the items that are part of the public interface
  `export declaration;` OR
  `export { declarations... };`

In it’s core, the module TS is rather simple:

- A file can declare itself as a module with the `module MODULE_NAME;` syntax.
- A module explicitly exports the items that are part of the public interface
- To get access to the exported elements you just import the module `import MODULE_NAME;`

In it’s core, the module TS is rather simple:

- A file can declare itself as a module with the `module MODULE_NAME;` syntax.
- A module explicitly exports the items that are part of the public interface
- To get access to the exported elements you just import the module `import MODULE_NAME;`
- Will also have support for submodules and grouping of modules

Example

Listing 5: module_a.cc

```
module moduleA;
void foo(int) {}
export int fun(int){
  return 5;
}
```

Listing 6: module_b.cc

```
module bmodule;
export int foo(){
  return 6;
}
void fun() {}
```

Listing 7: module_user.cc

```
import bmodule;
import moduleA;
int main() {
    return fun(5) + foo();
}
```

Current problems

- Macros. According to the proposal, macros are never exported. This is a language extension and macros are still preprocessor. Lots of people want macro exports

Current problems

- Macros. According to the proposal, macros are never exported. This is a language extension and macros are still preprocessor. Lots of people want macro exports
- Tooling. At the moment, the standard is written for an abstract machine, this would add requirements for build tools (which we of course all have)

References

- Modules TS (standards text) N7420
- P0142 - A module system for C++

LINKÖPING UNIVERSITY

This proposal adds three new keywords; `co_await`, `co_yield` and `co_return`. A function having one of these is called a coroutine. The goal is to be able to pause the state of execution and get intermediate results. A simple example is a generator:

```
generator<int> get_value() {
    for (int i{}; i<10;++i)
        co_yield i;
}
```

The `co_await` keyword let us call another coroutine and pause execution until a value is provided by the function:

```
future<int> fun();
void foo()
{
    auto val = co_await fun();
}
```

`co_return` is used instead of `return` in a coroutine.

Opposition

The proposal also includes a big set of library extensions and some authors from google (Geoff Romer, James Dennett and Chandler Carrouth) believe that the interface is blunt and difficult to work with.

> Fundamentally, the Coroutines TS does not provide a direct and efficient model of hardware: the primitive objects and operations that are used to implement coroutines are hidden behind an abstraction boundary. – P1063

References

- N4402 - Resumable Functions
- Coroutines TS - N723
- StackOverflow - What are coroutines in C++20?
- P1063 - Core Coroutines

LINKÖPING UNIVERSITY

The main goal of the executors proposal is to standardize a common way of controlling execution in C++.

Listing 8: Current way of woking with different execution models

```
void parallel_for(int facility, int n, function<void(int)> f) {
  if(facility == OPENMP) {
    #pragma omp parallel for
    for(int i = 0; i < n; ++i) {
      f(i);
    }
  }
  else if(facility == GPU) {
    parallel_for_gpu_kernel<<<n>>>(f);
  }
  else if(facility == THREAD_POOL) {
    global_thread_pool_variable.submit(n, f);
  }
}
```

The goal is to add a common abstraction for all execution models - an executor that controls execution.

```
// initialize executor somehow (depending on model)
my_executor_type my_executor = ...
// execute parallel for_each "on" my_executor
std::for_each(std::execution::par.on(my_executor),
              begin(data), end(data), func);
```

References

- P0761 - Executors Design Document
- P0443 - A unified Executors Proposal for C++ (standards text)
- Prototype implementation

LINKÖPING UNIVERSITY

The networking TS wants to add basic networking facilities to the standard library. It will be a variant of Boost.Asio.

What to expect

- TCP and UDP
- Support for client and server application code
- Scalability to handle multiple concurrent connections
- IPv4 and IPv6
- DNS
- timers

What Not to expect

- Protocols such as HTTP, SMTP, FTP, etc.
- Encryption (SSL,TLS)

References

- N4478 - Networking Library Proposal
- N4734 - Networking TS
- Example implementation on github
- Boost.Asio

LINKÖPING UNIVERSITY

The transactional memory proposal contains two transaction-safe blocks; synchronized blocks and atomic blocks. The main goal is to make it easier to work with synchronization between threads.

Synchronized blocks

A synchronized block works as if it locks a global mutex before entering the block and unlocks it afterwards. No thread will execute a synchronized block at the same time as any other thread. It will work as if they try to lock the same mutex, no matter which synchronized block.

- `synchronized { body }`

Synchronized blocks

A synchronized block works as if it locks a global mutex before entering the block and unlocks it afterwards. No thread will execute a synchronized block at the same time as any other thread. It will work as if they try to lock the same mutex, no matter which synchronized block.

- `synchronized { body }`
- There is also a proposed version where a domain can be specified so that each domain share one mutex (instead of using one global).
  `synchronized (domain) { body }`

Atomic blocks

An atomic block works as if all statements in that block is executed simultaneously, no other thread sees any intermediate state inside the atomic block. Three versions that handle exceptions differently.

- `atomic.noexcept { body }` UB if an exception is thrown.

Atomic blocks

An atomic block works as if all statements in that block is executed simultaneously, no other thread sees any intermediate state inside the atomic block. Three versions that handle exceptions differently.

- `atomic.noexcept { body }` UB if an exception is thrown.
- `atomic.commit { body }` If an exception is thrown, the transaction so far is committed and the exception is thrown.

Atomic blocks

An atomic block works as if all statements in that block is executed simultaneously, no other thread sees any intermediate state inside the atomic block. Three versions that handle exceptions differently.

- `atomic.noexcept { body }` UB if an exception is thrown.
- `atomic.commit { body }` If an exception is thrown, the transaction so far is committed and the exception is thrown.
- `atomic.cancel { body }` The transaction is aborted if exception is thrown and exception is thrown.

References

- N3919 - Transactional Memory Support for C++

LINKÖPING UNIVERSITY

A tool to let the compiler generate metaobjects to let us reason about static types. Will let us handle the following[2]:

- Data members. e.g. walking through the data members of a class
- Member types. e.g. walking through nested types or typedefs in a class
- Enumerators. The ability to, for example, make one-line serialization routines for enums
- Template instantiations. The ability to reflect on instantiated templates, such as `std::vector<int>`
- Alias support. The ability to distinguish between a typedef and its underlying type

[2]List taken verbatim from P0578

What does it involve?

- A new ”operator”, `$reflect` to generate metaobjects
- Library support to query metaobjects

```cpp
template <typename T>
T min(const T& a, const T& b) {
    log()   << "min<"
            << get_base_name_v<$reflect(T)>
            << ">(" << a << ", " << b << ") = ";
    T result = a<b?a:b;
    log()   << result << std::endl;
    return result;
}
```

References

- P0194 - Static reflection
- P0578 - Static Reflection in a Nutshell
- P0385 - Rationale and Design

LINKÖPING UNIVERSITY

Herb Sutter builds upon reflections and extends it alot to let us iterate over a class' members, query them, modify them and possibly add more. This will let us:

- Add abstractions
- Let the compiler enforce ”rules” such as how to create a good interface instead of relying on programmers to memorize requirements
- Letting libraries adding specialized types instead of requiring modifications to the language
- Eliminate the need to invent specific dialects or side-languages and specific compilers such as Qt Moc or C++/CX.

This is a very early version and could maybe be in C++23, but I'm pessimistic. All syntax in the examples below are very likely to change (examples lifted verbatim from the proposal).

Listing 9: pseduocode example

```
constexpr void interface(meta::type target, const meta::type source) {
    // - apply the "public" and "virtual" keywords to all member functions
    // - require that all member functions are public and virtual
    // - require no data members, copy or move functions
    // - generate a pure virtual destructor (if not user-supplied)
};
interface Shape {
  int area() const;
  void scale_by(double);
};
```

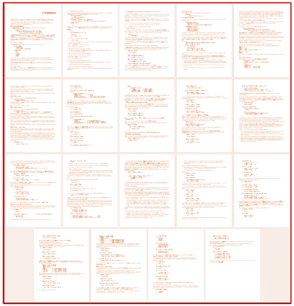
Listing 10: Generated class

```
class Shape {
public:
  virtual int area() const = 0;
  virtual void scale_by(double) = 0;
  virtual ~Shape() = 0;
```

Adding interface as a library vs as part of standard (C#)

Writing **as-if a new ‘language’ feature** using compile-time code + adding expressive power = XXL improvement

```
// C# language spec: ~20 pages of nontestable English
```



```
// User code (today’s Java or C#)
interface Shape {
    int area();
    void scale_by(double factor);
}
```

```
// (Proposed) C++ library: ~10 lines of testable code
constexpr void
interface(meta::type target, const meta::type source) {
    compiler.require(source.variables().empty(),
        "interfaces may not contain data");
    for (auto f : source.functions()) {
        compiler.require(!f.is_copy() && !f.is_move(),
            "interfaces may not copy or move; consider"
            " a virtual clone() instead");
        if (!f.has_access()) f.make_public();
        compiler.require(f.is_public(),
            "interface functions must be public");
        f.make_pure_virtual();
        ->(target) f;
    }
    ->(target) { virtual ~(source.name()$)() noexcept {} }
};
```

```
// User code (proposed C++)
interface Shape {
    int area() const;
    void scale_by(double factor);
};
```

Programming with QT

Compilation with qt

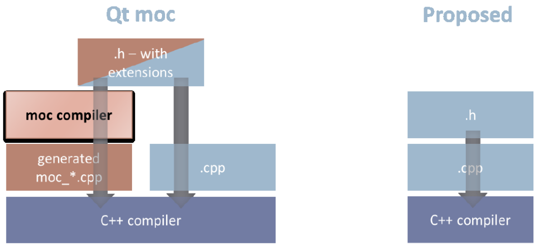


Figure 2: Qt ***extended*** *language +* ***side*** *compiler – build model vs. this proposal*

Programming with QT

Modified code

Qt moc style

```
class MyClass : public QObject {
    Q_OBJECT
public:
    MyClass( QObject* parent = 0 );
    Q_PROPERTY(int value READ get_value WRITE set_value)
    int  get_value() const { return value; }
    void set_value(int v)  { value = v; }
private:
    int value;
signals:
    void mySignal();
public slots:
    void mySlot();
};
```

This paper (proposed)

```
QClass MyClass {
    property<int> value { };
    signal mySignal();
    slot mySlot();
};
```

References

- P0707 - Metaclasses: Generative C++

LINKÖPING UNIVERSITY

Unicode

C++11 added support for string literals encoded in UTF-8, UTF-16 and UTF-32. It also added types to handle codepoints in UTF-16 and UTF-32 (`char16_t` and `char32_t`) but not for UTF-8. C++17 added string literals for UTF-8 but it's still stored as `char`!

```
auto str = u8"\0123";
```

`str` should be one code point, but since it's stored in a char array, it will be encoded as two chars (0xC4 and 0xA3).

Proposal P0482 wants to add a new type, `char8_t`, to represent a code point encoded in UTF-8. It also recommends the following:

- specialize `std::basic_string` and `std::string_view` for this type.
- An u8 string literal should generate a `char8_t` array
- Overload `operator<<` and `operator>>`

References

- P0482 - `char8_t`: A type for UTF-8 characters and strings

www.liu.se

LINKÖPING
UNIVERSITY