Doc No: Pxxxx DRAFT 1
Project: Programming Language - C++ - WG21
Audience: SG21 Contracts
Author: Andrew Tomazos <andrewtomazos@gmail.com>
Date: 2019-07-30
Vehicle: C++23

Proposal of Contract Primitives

1. Abstract

Instead of standardizing a contract system in C++23, we propose standardizing a set of language extensions in C++23 that will enable contract library designers to create better contract systems as non-macro pure C++ libraries. (We can then later standardize, on the basis of existing practice, one of these future user libraries in the C++26 or C++29 standard library). We call these proposed extensions collectively the *contract primitives*. The three contract primitives are **std::expression**, **boundary functions** and **return expressions**.

2. std::expression

2.1. Synopsis

```
namespace std {

template<typename T>
class expression {
public:
    expression(T) /*see below*/;

    T evaluate();

    void assume();

    const char* stringize();

    std::source_location location();

private:
```

```
    /*impl-defined*/
};

} // namespace std
```

2.2. Behaviour

The purpose of std::expression is to capture an expression without evaluating it immediately.

If an expression is a call to the std::expression<T>::expression(T) constructor, the argument subexpression of that expression is unevaluated.

Example 1

```
int f() { std::cout << "Hello, World!"; return 42; }

int main() {
    std::expression<int> expr = f(); // prints nothing
}
```

Conceptually the value of a std::expression initialized via std::expression<T>::expression(T) is the expression itself and not the result of its evaluation.

For an expression E that is the argument to std::expression<T>::expression(T), it is like if E is replaced by the tokens **[&]{ return E; }** :

That is:

```
std::expression<T> e = E;
```

...behaves something like:

```
std::function<T()> e = [&] { return E; }
```

A call to the .evaluate() member function of std::expression executes an evaluation of the expression:

Example 2

```
int f() { std::cout << "Hello, World!"; return 42; }

int main() {
```

```
    std::expression<int> expr = f(); // prints nothing
    int x = expr.evaluate(); prints "Hello, World!"
    return x; // returns 42
}
```

The purpose of the .assume() member function is to create an assumption that can be used for optimization, similar to __builtin_assume

A call to the .assume() member function of a std::expression<bool> has no side effects. In particular, it does not evaluate the expression. If not for this rule and if immediately after any would-be function call to the .assume() member function, a call to the .evaluate() member function were to be injected and executed - and the result of that hypothetical injected function call would have been anything other than it returned the value true - the entire program has undefined behaviour.

Example 3

```
bool hello(int argc) {
    cout << "hello";
    return argc == 2;
}

int main(int argc, char** argv) {
    if (argc != 2)
        std::cout << "wrong";
    std::expression<bool>(hello(argc)).assume();
}
```

Example 3 program run as:

```
$ ex3 arg1   # argc is 2
```

...has defined behaviour, and does **not** print "hello".

Example 3 program run as:

```
$ ex3 arg1 arg2    # argc is 3
```

...has entirely undefined behaviour, and may exhibit any behaviour from entry. In particular, it may or **may not** print "wrong" - even though that call appears before the bad assumption at runtime.

A call to .stringize() results in a const char* to a NTBS holding the source code of the expression.

Example 4

```
int main() {
    std::expression<int> e = 4 + 3;

    std::cout << e.stringize(); // prints "4 + 3"
}
```

A call to .location() returns the std::source_location of the expression that initialized the std::expression.

3. Boundary Functions

3.1 Syntax

We introduce two new keywords: **prologue** and **epilogue**

They are decl-specifiers:

decl-specifier:
- storage-class-specifier
- defining-type-specifier
- function-specifier
- friend
- typedef
- constexpr
- consteval
- inline
- **prologue**
- **epilogue**

We introduce the ability to place a function call expression in attribute position:

attribute-specifier:
- [[attribute-using-prefix$_{opt}$]]
- alignment-specifier
- **boundary-function-call**

boundary-function-call:
id-expression (expression-list$_{opt}$)

We also introduce return as an expression:

primary-expression:
literal
this
(expression)
id-expression
lambda-expression
fold-expression
requires-expression
return-expression

return-expression:
return

3.2 Semantics

The decl-specifiers **prologue** or **epilogue** shall only appear in the decl-specifier-seq of a function declaration of a namespace-scope function that returns void. At most one of the two may appear.

Example 5

```
prologue void p() {} // OK
epilogue void e() {} // OK
prologue int p2() { return 42; } // ill-formed
epilogue int e2() { return 42; } // ill-formed
prologue epilogue void f() {}  // ill-formed
```

A function marked **prologue** is a *prologue function*.

A function marked **epilogue** is an *epilogue function*.

A *boundary function* is a function that is either a prologue function or an epilogue function.

A function call expression to a boundary function may be placed as at attribute position on a function declaration.

Example 6

```
prologue void p() {}
epilogue void e() {}
void f1() p() {} // OK
void f2() e() {} // OK
void f3() p() e() {} // OK
```

Only the first declaration of a function within a TU may contain boundary function calls. The boundary function call set for a function must be the same across different translation units (much like the definition of an inline function).

Example 7

```
prologue void p() {} // OK
epilogue void e() {} // OK

void f1() p(); // OK
void f1() {} // OK

void f2() p();
void f2() e(); // ill-formed

void f3();
void f3() p() e() {} // ill-formed
```

A function definition of the form:

F { B }

that has:

- declared prologue function calls P1, P2, …., Pn
- declared epilogue function calls E1, E2, …, Em
- and for some unique identifier R that doesn't otherwise appear in the program

is equivalent to:

```
F {
   P1;
   P2;
   .
   .
  Pn;
  auto&& R = [&]{ B }();
```

```
    E1;
    E2;
    .
    .
    Em;
    return R;
}
```

Example 8

```
prologue void p() { cout << "p"; }
epilogue void e() { cout << "e"; }

int f() p() e() { cout << "f"; return 42; } // prints p f e

// f is equivalent to:

int f() {
    p();
    auto&& result = [&]{ cout << "f"; }();
    e();
    return result;
}
```

A return-expression provides a way to refer to the return value of the subject function in an epilogue function call.

A return-expression shall only appear within an epilogue function call at attribute position. It is equivalent to the id-expression R (the unique identifier from above).

Example 9

```
template<typename T>
epilogue void dumpres(T t) { std::cout << t; }

int f() dumpres(return) { return 42; }

int main() { f(); } // prints 42
```

(In the event of a syntactic ambiguity between a return statement and a return expression, the ambiguity is resolved in favor of the return statement.)

4. Example Applications

To demonstrate the utility of the proposed contract primitives, we show some examples of possible contract systems that use them.

4.1 No Semantic Contract System

First we present a contract system that performs no runtime evaluation of its predicates:

```
namespace cs41 {

inline void assert(std::expression<bool>) {}
prologue inline void precond(std::expression<bool>) {}
epilogue inline void postcond(std::expression<bool>) {}

}  // namespace cs41
```

Example 10

```
void p(int x) { std::cout << x; return x; }

int f(int x)
    cs41::precond(p(x) > 3)
    cs41::postcond(return > p(10));

int f(int x)
{
    return x + 3;
}

int main() {
    f(10); // OK: prints nothing
    f(5); // OK: prints nothing
    f(2); // OK: prints nothing
    return 42; // reached
}
```

A contract system like this could be used purely by static analysis tools and documentation generators - but has no run-time effect.

4.2 Runtime Checking Contract System

We present a contract library that unconditionally evaluates and checks the predicate, and aborts the program if false:

```
namespace cs42 {

    inline void assert(bool predicate) {
        if (!predicate) std::abort();
    }

    prologue inline void precond(bool predicate) {
        if (!predicate) std::abort();
    }

    epilogue inline void postcond(bool predicate) {
        if (!predicate) std::abort();
    }

}  // namespace cs42
```

Example 11

```
float square_area(float width)
   cs42::precond(width >= 0)
   cs42::postcond(return >= 0);

float square_area(float width) {
   return width*width;
}

int main() {
   square_area(3.0); // OK
   square_area(-3.0); // ABORT
   cs41::assert(false); // ABORT
}
```

4.3 Modal Contract System

We present a contract system with three runtime modes: ignore, warn and error. This mode could be set via a command-line argument, and also altered during runtime.

In ignore mode the predicates are unevaluated.

In warn mode the predicates are checked and a warning is logged if false, the program continues.

In error mode the predicates are checked and the program aborts if false.

```
namespace cs43 {

    enum Mode { ignore, warn, error };
    inline Mode& mode() { static Mode m = ignore; return m; }

    inline void process_predicate(std::expression<bool> predicate) {
        if (mode() == ignore) return;

        if (predicate.evaluate()) return;

        std::cerr <<
           "contract " << predicate.stringize() << " failed\n";

        if (mode() == error) std::abort();
    }

    inline void assert(std::exception<bool> predicate) {
        process_predicate(predicate);
    }

    prologue inline void precond(std::exception<bool> predicate) {
        process_predicate(predicate);
    }

    epilogue inline void postcond(std::exception<bool> predicate) {
        process_predicate(predicate);
    }

}  // namespace cs43
```

Example 12

```
int f(int x)
```

```
    cs43::precond(x > 3)
    cs43::postcond(return < 9);

int f(int x) { return 2*x; }

int main(int argc, char** argv) {
    // set contract mode from command-line
    std::string mode = argv[1];
    if (mode == "ignore")
        cs43::mode() = cs43::ignore;
    else if (mode == "warn")
        cs43::mode() = cs43::warn;
    else
        cs43::mode() = cs43::error;


  f(4); // OK
  f(2); // violation: behaviour depends on argv[1]
  f(5); // violation: behaviour depends on argv[1]
}
```

4.4 Elaborate Error Message Contract System

We show a contract system that prints the kind, text of the predicate, and the origin file and line on a violation...

```
namespace cs44 {

void print_error(const char* kind, std::expression<bool> predicate) {
    std::cerr << "contract violation\n";
    std::cerr << "kind: " << kind << "\n";
    std::cerr << "predicate: " << predicate.stringize() << "\n";
    std::cerr << "file: " << predicate.location().file_name();
    std::cerr << "line: " << predicate.location().line();
}

inline void assert(std::exception<bool> predicate) {
    if (!predicate) print_error("assert", predicate);
}

prologue inline void precond(bool predicate) {
```

```
    if (!predicate) print_error("predcondition", predicate);
}

epilogue inline void postcond(bool predicate) {
    if (!predicate) print_error("postcondition", predicate);
}

} // namespace cs44
```

4.5 N4820 WP Contracts System

We show a pure library contracts system built atop the proposed contract primitives, that has the feature set of the contract system in the pre-Cologne June 2019 Working Paper N4820.

```
namespace cs45 {

enum class BuildMode {
    off,
    default_,
    audit
};

constexpr BuildMode build_mode =
#ifdef CS45_BUILD_MODE
    CS45_BUILD_MODE;
#else
    default_;
#endif

enum class ContinuationMode {
    on,
    off
};

Constexpr ContinuationMode continuation_mode =
#ifdef CS45_CONTINUATION_MODE
    CS45_CONTINUATION_MODE
#else
    off
#endif

enum class ContractLevel {
```

```
    default_,
    audit,
    axiom,
}

class ContractViolation {
Public:
ContractViolation(uint_least32_t line_number,
                  const std::string file_name,
                  const std::string function_name,
                  const std::string comment,
                  const std::string assertion_level) :
    line_number_(line_number),
    file_name_(file_name),
    function_name_(function_name),
    comment_(comment),
    assertion_level_(assertion_level)

    uint_least32_t line_number() const noexcept
    { return line_number_; }

    const char* file_name() const noexcept
    { return file_name_.c_str(); }

    string_view function_name() const noexcept
    { return function_name_; }

    string_view comment() const noexcept
    { return comment_; }

    string_view assertion_level() const noexcept
    { return assertion_level_; }

private:
    uint_least32_t line_number_
    std::string file_name_;
    std::string function_name_;
    std::string comment_;
    std::string assertion_level_;
};

using ContractViolationHandler = void(*)(const ContractViolation&);
```

```
extern ContractViolationHandler contract_violation_handler;

template<ContractLevel contract_level>
Inline void process_contract(
    std::expression<bool> predicate) {
    if constexpr (build_mode == BuildMode::default_ &&
                  contract_level == ContractLevel::axiom) {
        predicate.assume();
    }
    else if constexpr (
            build_mode != BuildMode::off
            && contract_level != ContractLevel::axiom)
            && (contract_level == ContractLevel::default_ ||
                build_mode == BuildMode::audit_)
    {
        if (predicate.evaluate())
            return;

        ContractViolation violation(
            predicate.location().line_number(),
            predicate.location().file_name(),
            predicate.location().function_name(),
            predicate.stringize(),
            Contract_level == ContractLevel::axiom ? "axiom" :
            Contract_level == ContractLevel::default_ ? "default" :
            Contract_level == ContractLevel::audit ? "audit" : ""
        );

        if (contract_violation_handler)
             contract_violation_handler(violation);

        If (continuation_mode == ContinuationMode::off)
            std::abort();
    }
}

inline void assert(std::expression<bool> predicate) {
    process_contract<ContractLevel::default_>(predicate);
}

inline void assert_audit(std::expression<bool> predicate) {
    process_contract<ContractLevel::audit>(predicate);
}
```

```
inline void assert_axiom(std::expression<bool> predicate) {
    process_contract<ContractLevel::axiom>(predicate);
}

prologue inline void expects(std::expression<bool> predicate) {
    process_contract<ContractLevel::default_>(predicate);
}

prologue inline void expects_audit(std::expression<bool> predicate) {
    process_contract<ContractLevel::audit>(predicate);
}

prologue inline void expects_axiom(std::expression<bool> predicate) {
    process_contract<ContractLevel::axiom>(predicate);
}

epilogue inline void ensures(std::expression<bool> predicate) {
    process_contract<ContractLevel::default_>(predicate);
}

epilogue inline void ensures_audit(std::expression<bool> predicate) {
    process_contract<ContractLevel::audit>(predicate);
}

epilogue inline void ensures_axiom(std::expression<bool> predicate) {
    process_contract<ContractLevel::axiom>(predicate);
}

}  // namespace cs45
```