Contracts programming for C++20

Current proposal status

J. Daniel Garcia

ARCOS Group
University Carlos III of Madrid
Spain

February 27, 2018

Warning

Who am I?

- A C++ programmer.
  - Started writing C++ code in 1989.

Who am I?

- A C++ programmer.
  - Started writing C++ code in 1989.
- A university professor in Computer Architecture.

Who am I?

- A C++ programmer.
  - Started writing C++ code in 1989.
- A university professor in Computer Architecture.
- A ISO C++ language standards committee member.

Who am I?

- A C++ programmer.
  - Started writing C++ code in 1989.
- A university professor in Computer Architecture.
- A ISO C++ language standards committee member.
- **My goal**: Improve applications programming.
  - **Performance** → **faster** applications.
  - **Energy efficiency** → **better** performance per Watt.
  - **Maintainability** → **easier** to modify.
  - **Reliability** → **safer** components.

ARCOS@uc3m

- **UC3M**: A young international research oriented university.
- **ARCOS**: An applied research group.
  - **Lines**: High Performance Computing, Big data, Cyberphysical systems, **Programming Models for Applications Improvement**.
- **Improving applications**:
  - **REPARA**: Reengineering and Enabling Performance and poweR of Applications. FP7-ICT (2013–2016).
  - **RePhrase**: REfactoring Parallel Heterogeneous Resource Aware Applications. H2020-ICT (2015–2018).
  - **ASPIDE**: exAScale ProgrammIng models for extreme Data procEssing. H2020-FET-HPC (2018–2020).
- **Standardization**:
  - ISO/IEC JTC/SC22/WG21. ISO C++ Committee.

Why are we here?

- Because we are concerned about writing correct software.

Why are we here?

- Because we are concerned about writing correct software.
- Isn't a library solution enough?

Why are we here?

- Because we are concerned about writing correct software.

- Isn't a library solution enough?
  - We already tried that!
  - Compilers and static analyzers do not understand that approach.

Why are we here?

- Because we are concerned about writing correct software.
- Isn't a library solution enough?
  - We already tried that!
  - Compilers and static analyzers do not understand that approach.
- What did others do?

Why are we here?

- Because we are concerned about writing correct software.
- Isn't a library solution enough?
  - We already tried that!
  - Compilers and static analyzers do not understand that approach.
- What did others do?
  - Several language solutions out there (D, Ada, C#).

Contracts in C++

- First proposal for contracts programming in 2005.
  - **N1613**: Proposal to add Design by Contract to C++. Throsten Ottosen.
  - Died during the C++0x process.

Contracts in C++

- First proposal for contracts programming in 2005.
  - **N1613**: Proposal to add Design by Contract to C++. Throsten Ottosen.
  - Died during the C++0x process.

- Next attempt in 2013.
  - **N3604**: Centralized Defensive-Programming Support for Narrow Contracts. John Lakos, Alexei Zakharov.

Current contracts effort

- 2014:Multiple proposals on contracts programming.
  - Discussions in the standards committee.

Current contracts effort

- 2014:Multiple proposals on contracts programming.
  - Discussions in the standards committee.
- 2016: Joint proposal trying to consider trade-offs.
  - Gabriel Dos Reis, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, Bjarne Stroustrup.
  - Targeting C++20.

Correctness and Robustness

- In the design of a library there is a tension between two related properties: **robustness** and **correctness**.

Correctness and Robustness

- In the design of a library there is a tension between two related properties: **robustness** and **correctness**.
  - **Correctness** → Degree to which a software component matches its specification.

Correctness and Robustness

- In the design of a library there is a tension between two related properties: **robustness** and **correctness**.
  - **Correctness** $\rightarrow$ Degree to which a software component matches its specification.
  - **Robustness** $\rightarrow$ Ability of a software component to react appropriately to abnormal conditions.

Correctness and Robustness

- In the design of a library there is a tension between two related properties: **robustness** and **correctness**.
  - **Correctness** → Degree to which a software component matches its specification.
  - **Robustness** → Ability of a software component to react appropriately to abnormal conditions.
- Today many libraries use a single feature for managing both properties: **exception handling**.
  - **We need to revisit this approach!**

Exceptions in use

- When a failure happens, we use exceptions as an error reporting mechanism.
  - Notify that an error has occurred and needs to be handled.
  - We decouple error identification from error handling.
  - **Example**: Throwing **bad_alloc**.

Exceptions in use

- When a failure happens, we use exceptions as an error reporting mechanism.
  - Notify that an error has occurred and needs to be handled.
  - We decouple error identification from error handling.
  - **Example**: Throwing **bad_alloc**.
- When library detects an assumption was not met, it needs a mechanism to react.
  - Assumption not met $\Rightarrow$ **contract violation**.
  - What do we do on contract violations?
    - Ignore reality $\rightarrow$ **Do not call me!**
    - Document.
    - Throw exceptions.

Exceptions in use

- When a failure happens, we use exceptions as an error reporting mechanism.
  - Notify that an error has occurred and needs to be handled.
  - We decouple error identification from error handling.
  - **Example**: Throwing **bad_alloc**.
- When library detects an assumption was not met, it needs a mechanism to react.
  - Assumption not met $\Rightarrow$ **contract violation**.
  - What do we do on contract violations?
    - Ignore reality $\rightarrow$ **Do not call me!**
    - Document.
    - Throw exceptions.
- **Robustness** and **correctness** are orthogonal properties and **should be managed independently**.

Robustness in the C++ standard library

- **Robustness**: Identification and handling of abnormal situations.
  - Those situations occur in completely correct programs.
  - **Example**: Failure to allocate memory.
  - Is end of file a robustness issue?

Robustness in the C++ standard library

- **Robustness**: Identification and handling of abnormal situations.
  - Those situations occur in completely correct programs.
  - **Example**: Failure to allocate memory.
  - Is end of file a robustness issue?

- The C++ standard library identifies those cases by specifying
  - i the condition firing the situation.
  - ii the exception that will be thrown to notify.

Robustness in the C++ standard library

- **Robustness**: Identification and handling of abnormal situations.
  - Those situations occur in completely correct programs.
  - **Example**: Failure to allocate memory.
  - Is end of file a robustness issue?

- The C++ standard library identifies those cases by specifying
  i. the condition firing the situation.
  ii. the exception that will be thrown to notify.

- **T * allocator<T>::allocate(std::size_t n);**

  Throws: **bad_alloc** if storage cannot be obtained.

Correctness and contracts

- **Correctness** → Finding programming errors.
  - Yes! Sometimes we write incorrect software.

Correctness and contracts

- **Correctness** → Finding programming errors.
  - Yes! Sometimes we write incorrect software.
- Who's guilty?
- A contract violation happens because:
  - A caller does not fulfil the expectations before calling a function.
  - A callee does not fulfill what should be ensured after its own execution.

Correctness and contracts

- **Correctness** → Finding programming errors.
  - Yes! Sometimes we write incorrect software.
- Who's guilty?
- A contract violation happens because:
  - A caller does not fulfil the expectations before calling a function.
  - A callee does not fulfill what should be ensured after its own execution.
- A key difference:
  - A program failure is usually due to external conditions and cannot be avoided.
  - A contract violation ***should*** never happen in a correct program.

Correctness in the C++ standard library

- From the standard:

> Violation of the preconditions specified in a function's *Requires*: paragraph results in undefined behavior unless the functions *Throws*: paragraph specifies throwing an exception when the precondition is violated.

Correctness in the C++ standard library

- From the standard:

> Violation of the preconditions specified in a function's *Requires*: paragraph results in undefined behavior unless the functions *Throws*: paragraph specifies throwing an exception when the precondition is violated.

- In practice, there are two approaches in the standard library:
  - Do nothing → **Undefined behaviour**.
  - Notify → **Throw an exception**.

What is a contract?

- A contract is the set of **preconditions**, **postconditions** and **assertions** associated to a function.

What is a contract?

- A contract is the set of **preconditions**, **postconditions** and **assertions** associated to a function.
  - **Precondition**: What are the *expectations* of the function?

What is a contract?

- A contract is the set of **preconditions**, **postconditions** and **assertions** associated to a function.
  - **Precondition**: What are the *expectations* of the function?
  - **Postconditions**: What must the function *ensure* upon termination?

What is a contract?

- A contract is the set of **preconditions**, **postconditions** and **assertions** associated to a function.
  - **Precondition**: What are the *expectations* of the function?
  - **Postconditions**: What must the function *ensure* upon termination?
  - **Assertions**: What predicates must be satisfied in specific locations of a function body?

Expectations

- **Precondition**
  - A predicate that should hold upon entry into a function.
  - It expresses a function's expectation on its arguments and/or the state of objects that may be used by the function.
  - Expressed by attribute **expects**.

Expectations

- **Precondition**
  - A predicate that should hold upon entry into a function.
  - It expresses a function's expectation on its arguments and/or the state of objects that may be used by the function.
  - Expressed by attribute **expects**.

```
double sqrt(double x) [[expects: x>0]];
```

Expectations

- **Precondition**
  - A predicate that should hold upon entry into a function.
  - It expresses a function's expectation on its arguments and/or the state of objects that may be used by the function.
  - Expressed by attribute **expects**.

```
double sqrt(double x) [[expects: x>0]];

class queue {
  // ...
  void push(const T & x) [[expects: ! full () ]];
  // ...
};
```

Expectations

- **Precondition**
  - A predicate that should hold upon entry into a function.
  - It expresses a function's expectation on its arguments and/or the state of objects that may be used by the function.
  - Expressed by attribute **expects**.

```
double sqrt(double x) [[expects: x>0]];

class queue {
  // ...
  void push(const T & x) [[expects: ! full () ]];
  // ...
};
```

- Preconditions use a modified attribute syntax.
- The expectation is part of the function declaration.

Assurances

- **Postcondition**
  - A predicate that should hold upon exit from a function.
  - It expresses the conditions that a function should ensure for the return value and/or the state of objects that may be used by the function.
  - Postconditions are expressed by **ensures** attributes.

Assurances

- **Postcondition**
  - A predicate that should hold upon exit from a function.
  - It expresses the conditions that a function should ensure for the return value and/or the state of objects that may be used by the function.
  - Postconditions are expressed by **ensures** attributes.

```
double sqrt(double x)
  [[expects: x>=0]]
  [[ensures result:  result >=0]];
```

Assurances

- **Postcondition**
  - A predicate that should hold upon exit from a function.
  - It expresses the conditions that a function should ensure for the return value and/or the state of objects that may be used by the function.
  - Postconditions are expressed by **ensures** attributes.

```
double sqrt(double x)
  [[expects: x>=0]]
  [[ensures result: result>=0]];
```

- Postconditions may introduce a name for the result of the function.

Assertions

- **Assertions**
  - A predicate that should hold at its point in a function body.
  - It expresses the conditions that must be satisfied, on objects that are accessible at its point in a body.
  - Assertions are expressed by assert attributes.

Assertions

- **Assertions**
  - A predicate that should hold at its point in a function body.
  - It expresses the conditions that must be satisfied, on objects that are accessible at its point in a body.
  - Assertions are expressed by assert attributes.

```
double add_distances(const std::vector<double> & v)
  [[expects r: r>=0.0]]
{
  double r = 0.0;
  for (auto x : v) {
    [[assert: x >= 0.0]];
    r += x;
  }
  return r;
}
```

Effect of contracts

- A contract has no observable effect on a correct program (except performance).
  - The only semantic effect of a contract happens if it is violated.

Effect of contracts

- A contract has no observable effect on a correct program (except performance).
  - The only semantic effect of a contract happens if it is violated.
- Why do we use attributes syntax?
  - Contract may be checked or not.
  - Attributes are not part of function type.
  - However, **contracts are not an optional feature**.

Effect of contracts

- A contract has no observable effect on a correct program (except performance).
  - The only semantic effect of a contract happens if it is violated.
- Why do we use attributes syntax?
  - Contract may be checked or not.
  - Attributes are not part of function type.
  - However, **contracts are not an optional feature**.
- Contracts checking and corresponding effects depend on build system settings.
  - Default: Contract violation $\Rightarrow$ Program termination.

Repeating a contract

- Any redeclaration of a function has either the same contract or completely omits the contract.

Repeating a contract

- Any redeclaration of a function has either the same contract or completely omits the contract.

```
int f(int x)
  [[expects: x>0]]
  [[ensures r: r >0]];

int f (int x) ; // OK. No contract.

int f ( int x)
  [[expects: x>=0]]; // Error missing ensures and different expects

int f(int x)
  [[expects: x>0]]
  [[ensures r: r >0]]; //OK. Same contract.
```

Repeating a contract

- But argument names may differ.

Repeating a contract

- But argument names may differ.

```
int f(int x)
  [[expects: x>0]]
  [[ensures r: r >0]];

int f(int y)
  [[expects: y>0]]
  [[ensures z: z >0]];
```

Assertion level

- Every contract expression has an associated *assertion level*.

Assertion level

- Every contract expression has an associated *assertion level*.
- Contract levels: **always**, **default**, **audit**, **axiom**.
  - Checks will be effectively performed depending on build mode.

Assertion level

- Every contract expression has an associated *assertion level*.
- Contract levels: **always**, **default**, **audit**, **axiom**.
  - Checks will be effectively performed depending on build mode.
- Default level can be omitted.

```
void f(element & x) [[expects: x.valid() ]];
void g(element & x) [[expects default: x.valid() ]];
```

Assertion level

- Every contract expression has an associated *assertion level*.
- Contract levels: **always**, **default**, **audit**, **axiom**.
  - Checks will be effectively performed depending on build mode.
- Default level can be omitted.

```
void f(element & x) [[expects: x.valid() ]];
void g(element & x) [[expects default: x.valid() ]];
```

  - Cost of **default** checking is expected to be small compared to function execution.

Audit checks

- An **audit** *assertion level* is expected to be used in cases where the cost of a run-time check is assumed to be large compared to function execution.
  - Or at least significant.

```
template <typename It, typename T>
bool binary_search(It first, It last, const T & x)
  [[ expects audit: is_sorted(first, last) ]];
```

Axiom checks

- An **axiom** *assertion level* is expected to be used in cases where the run-time check will **never** be performed.
  - Still they need to be valid C++.
  - They are formal comments for humans and/or static analyzers.

Axiom checks

- An **axiom** *assertion level* is expected to be used in cases where the run-time check will **never** be performed.
  - Still they need to be valid C++.
  - They are formal comments for humans and/or static analyzers.

```
template <typename InputIterator>
InputIterator my_algorithm(InputIterator first, InputIterator last)
  [[expects axiom: first!=last && reachable(first, last)]];
```

Axiom checks

- An **axiom** *assertion level* is expected to be used in cases where the run-time check will **never** be performed.
  - Still they need to be valid C++.
  - They are formal comments for humans and/or static analyzers.

```
template <typename InputIterator>
InputIterator my_algorithm(InputIterator first, InputIterator last)
  [[expects axiom: first!=last && reachable(first,last)]];
```

- Axioms are not evaluated.
  - They may contain calls to declared but undefined functions.

Always checks

- An **always** *assertion level* is expected to be used in cases where the run-time check is so critical that it should never be switched off.
  - It should be used only exceptionally.

Build levels

- Every translation is performed in a *build level*:
  - **off**: No run-time checking is performed.
  - **default**: Checks with **default** levels are checked.
  - **audit**: Checks with **default** and **audit** levels are checked.

Build levels

- Every translation is performed in a *build level*:
  - **off**: No run-time checking is performed.
  - **default**: Checks with **default** levels are checked.
  - **audit**: Checks with **default** and **audit** levels are checked.

- How do you select the *build level*:
  - **No way of selecting in source code.**
  - An option from your compiler.

Contract checking

- If a function has multiple preconditions or postconditions that would be checked, their **evaluation will be performed in the order they appear**

Contract checking

- If a function has multiple preconditions or postconditions that would be checked, their **evaluation will be performed in the order they appear**

```
void f(int * p)
  [[expects: p!=nullptr]]
  [[expects: *p == 0]]  // Only checked when p!=nullptr
{
  *p = 1;
}
```

Contract violation handlers

- A translation unit has an associated contract violation handler.

Contract violation handlers

- A translation unit has an associated contract violation handler.
- A contract violation handler is the function to be called when a contract is broken.
  - Function with specific signature.

```
void (const std::contract_violation &);
```

Contract violation handlers

- A translation unit has an associated contract violation handler.
- A contract violation handler is the function to be called when a contract is broken.
  - Function with specific signature.

```
void (const std::contract_violation &);
```

- If you do not supply a handler, the default is **std::abort()**.

Contract violation handlers

- A translation unit has an associated contract violation handler.
- A contract violation handler is the function to be called when a contract is broken.
  - Function with specific signature.

```
void (const std::contract_violation &);
```

- If you do not supply a handler, the default is **std::abort()**.
- If you want to supply a handler:

Contract violation handlers

- A translation unit has an associated contract violation handler.
- A contract violation handler is the function to be called when a contract is broken.
  - Function with specific signature.

```
void (const std::contract_violation &);
```

- If you do not supply a handler, the default is **std::abort()**.
- If you want to supply a handler:
  - **No way of setting through source code**.

Contract violation handlers

- A translation unit has an associated contract violation handler.
- A contract violation handler is the function to be called when a contract is broken.
  - Function with specific signature.

```
void (const std::contract_violation &);
```

- If you do not supply a handler, the default is **std::abort()**.
- If you want to supply a handler:
  - **No way of setting through source code**.
  - **No way of asking which is current handler**.

Contract violation handlers

- A translation unit has an associated contract violation handler.
- A contract violation handler is the function to be called when a contract is broken.
  - Function with specific signature.

```
void (const std::contract_violation &);
```

- If you do not supply a handler, the default is **std::abort()**.
- If you want to supply a handler:
  - **No way of setting through source code**.
  - **No way of asking which is current handler**.
  - **An option in your compiler to supply it**.

Contract violation handlers

- A translation unit has an associated contract violation handler.
- A contract violation handler is the function to be called when a contract is broken.
  - Function with specific signature.

```
void (const std::contract_violation &);
```

- If you do not supply a handler, the default is **std::abort()**.
- If you want to supply a handler:
  - **No way of setting through source code**.
  - **No way of asking which is current handler**.
  - **An option in your compiler to supply it**.
  - **Security sensitive systems may prevent arbitrary handlers**.

Information for the handler

- Function with specific signature.

```
void (const std::contract_violation &);
```

Information for the handler

- Function with specific signature.

```
void (const std::contract_violation &);
```

- Minimum information inf **contract_violation**:

```
class contract_violation {
public:
  int line_number() const noexcept;
  string_view file_name() const noexcept;
  string_view function_name() const noexcept;
  string_view comment() const noexcept;
};
```

Information for the handler

- Function with specific signature.

```
void (const std:: contract_violation  &);
```

- Minimum information inf **contract_violation**:

```
class contract_violation  {
public:
   int  line_number() const noexcept;
   string_view  file_name()  const noexcept;
   string_view  function_name() const noexcept;
   string_view  comment() const noexcept;
};
```

- Might get simplified by **std::experimental::source_location**.

What happens after the violation handler?

- Two basic options:
  - Program *finishes* execution.
  - Program resumes execution.

What happens after the violation handler?

- Two basic options:
  - Program *finishes* execution.
  - Program resumes execution.

- An option in your compiler to select *continuation mode*:
  - **off**: Do not resume execution.
    - Default option.
  - **on**: Resume execution.

What happens after the violation handler?

- Two basic options:
  - Program *finishes* execution.
  - Program resumes execution.
- An option in your compiler to select *continuation mode*:
  - **off**: Do not resume execution.
    - Default option.
  - **on**: Resume execution.
- But remember:

What happens after the violation handler?

- Two basic options:
  - Program *finishes* execution.
  - Program resumes execution.
- An option in your compiler to select *continuation mode*:
  - **off**: Do not resume execution.
    - Default option.
  - **on**: Resume execution.
- But remember:
  - **No way of setting through source code**.

What happens after the violation handler?

- Two basic options:
  - Program *finishes* execution.
  - Program resumes execution.
- An option in your compiler to select *continuation mode*:
  - **off**: Do not resume execution.
    - Default option.
  - **on**: Resume execution.
- But remember:
  - **No way of setting through source code**.
  - **No way of asking which is current mode**.

Why do we want to continue?

- Gradual introduction of contracts.

Why do we want to continue?

- Gradual introduction of contracts.

- Testing the contracts themselves.

Why do we want to continue?

- Gradual introduction of contracts.
- Testing the contracts themselves.
- Plugin management.

Continuation mode and optimizations

- Assertion information may be used by optimizers.

```
[[assert: ptr!=nullptr]];
// ...
if (ptr!=nullptr) { // Can be optimized out
  do_stuff();
}
```

Continuation mode and optimizations

- Assertion information may be used by optimizers.

```
[[assert: ptr!=nullptr ]];
// ...
if (ptr!=nullptr) { // Can be optimized out
  do_stuff();
}
```

- If continuation mode is **off**, then **if** is never reached.

Continuation mode and optimizations

- Assertion information may be used by optimizers.

```
[[assert: ptr!=nullptr]];
// ...
if (ptr!=nullptr) { // Can be optimized out
  do_stuff();
}
```

- If continuation mode is **off**, then **if** is never reached.
- If continuation mode is **on**, then **if** would be reached.

Continuation mode and optimizations

- Assertion information may be used by optimizers.

```
[[assert: ptr!=nullptr]];
// ...
if (ptr!=nullptr) { // Can be optimized out
  do_stuff();
}
```

- If continuation mode is **off**, then **if** is never reached.
- If continuation mode is **on**, then **if** would be reached.
  - But the **if** might get **optimized out**!

Continuation mode and optimizations

- Assertion information may be used by optimizers.

```
[[assert: ptr!=nullptr]];
// ...
if (ptr!=nullptr) { // Can be optimized out
  do_stuff();
}
```

- If continuation mode is **off**, then **if** is never reached.
- If continuation mode is **on**, then **if** would be reached.
  - But the **if** might get **optimized out**!
  - **Continuation after violation is technically undefined behavior**.

Contracts and **noexcept**

- What happens to **noexcept** function if its contract is broken?
  - With continuation mode set to **off** program finishes.
  - With continuation mode set to **on** program resumes.
  - But, what if the handler throws an exception’
    - Program invokes **terminate()** *as-if* an exception was thrown inside functions.

```
void f(int x) [[expects: x > 0]];

void g() {
  f(−1); // Invokes terminate if handler throws
}
```

Repeating a contract

- Any redeclaration of a function has either the same contract or completely omits the contract.

Repeating a contract

- Any redeclaration of a function has either the same contract or completely omits the contract.

```
int f(int x)
  [[expects: x>0]]
  [[ensures r: r >0]];

int f (int x) ; // OK. No contract.

int f ( int x)
  [[expects: x>=0]]; // Error missing ensures and different expects

int f(int x)
  [[expects: x>0]]
  [[ensures r: r >0]]; //OK. Same contract.
```

Preconditions on functions

- The expression of a precondition from a function may use:
  - The function's arguments.
  - Any non-local object.

```
constexpr int max = 100;
std::string name{"Daniel"};

bool f(int x, std::string s)
  [[expects: x>0]]  // OK. x is an argument.
  [[expects: x<max]] // OK max is non-local
  [[expects: s.length()>0]]  // OK. s is an argument
  [[expects: s!=name]]; // OK. name is non-local
```

Preconditions on **constexpr** functions

- The expression of a precondition from a **constexpr** function may use:
  - The function's arguments.
  - Any non-local object that is **constexpr**.
  - **but it cannot access non-local objects that are not constexpr**.

```
constexpr int max = 100;
std::string name{"Daniel"};

constexpr bool f(int x, std::string s)
  [[expects: x>0]]  // OK. x is an argument.
  [[expects: x<max]] // OK max is constexpr
  [[expects: s.length()>0]]  // OK. s is an argument
  [[expects: s!=name]]; // Error name is a non-local variable
```

Modifications in contracts

- A program with a contract expression that performs an **observable modification** of an object is **ill-formed**.
  - Your compiler might give a diagnostic.

Modifications in contracts

- A program with a contract expression that performs an **observable modification** of an object is **ill-formed**.
  - Your compiler might give a diagnostic.

```
int f(int x)
  [[expects: x++ > 0]]      // Error
  [[ensures r: r == ++x]];  // Error
```

Modified arguments and postconditions

- If a postcondition uses an argument and the function body modifies that value, the program is ill-formed.

Modified arguments and postconditions

- If a postcondition uses an argument and the function body modifies that value, the program is ill-formed.

```
int f(int x)
  [[ensures r: r==x]
{
  return ++x; // Error x used in postcondition
}
```

Modified arguments and postconditions

- If a postcondition uses an argument and the function body modifies that value, the program is ill-formed.

```
int f(int x)
  [[ensures r: r==x]
{
  return ++x; // Error x used in postcondition
}
```

- Workaround:

```
int f(int x) {
  int oldx = x;
  auto r = ++x;
  [[assert: r==oldx]];
}
```

But you con modify pointer contents

- A pointer value is different from the pointed value.

But you con modify pointer contents

- A pointer value is different from the pointed value.

```
void f(int * ptr)
  [[ensures: ptr!=nullptr]]
{
  *ptr = 42
}
```

Contracts in templated function

- The expression of a contract from a function template or a member function of a class template may use the template arguments.

Contracts in templated function

- The expression of a contract from a function template or a member function of a class template may use the template arguments.

```
template <typename T, int size>
class table {
public:
  // ...
  T & operator[](int i)
    [[expects: 0<=i && i<size]];
};
```

Contracts and visibility

- The contract from a public function shall not use members from protected or private interfaces.
- The contract from a protected function shall not use members from private interface.

Contracts and visibility

- The contract from a public function shall not use members from protected or private interfaces.
- The contract from a protected function shall not use members from private interface.

```
template <typename T>
class table {
public:
  // ...
  T & operator[](int i)
    [[expects: 0<=i && i<size_]]; // Error. size_ is private

private:
  // ...
  int size_;
};
```

Contracts and function pointers

- A function pointer shall not include a contract.
- A call through a function pointer to functions with a contract shall perform contract assertions checking once.

Contracts and function pointers

- A function pointer shall not include a contract.
- A call through a function pointer to functions with a contract shall perform contract assertions checking once.

```
using fpt = int (*)(int x)
  [[expects: x>=0]]
  [[ensures r: r>0]]; // Error.

int g(int x) [[expects: x>=0]] [[ensures r: r>0]]
{
 return x+1;
}

int (*pf)(int) = g; // OK
```

Contracts and inheritance

- An **overriding function** shall have exactly the **same contract** that was declared for that function in the base class.
  - But the contract may be omitted in the overridden function.

Contracts and inheritance

- An **overriding function** shall have exactly the **same contract** that was declared for that function in the base class.
  - But the contract may be omitted in the overridden function.

```
struct B {
public:
  virtual void f (int x) [[expects: x>0]];
  // ...
};

struct D : public B {
public:
  virtual void f (int x) override; // OK. expects: x>0
  // ...
};
```

Contracts and inheritance

- An **overriding function** shall have exactly the **same contract** that was declared for that function in the base class.
  - Or it may be repeated.

Contracts and inheritance

- An **overriding function** shall have exactly the **same contract** that was declared for that function in the base class.
  - Or it may be repeated.

```
struct B {
public:
  virtual void f (int x) [[expects: x>0]];
  // ...
};

struct D : public B {
public:
  virtual void f (int x) override [[expects: x>0]]; // OK
  // ...
};
```

Contracts and inheritance

- An **overriding function** shall have exactly the **same contract** that was declared for that function in the base class.
  - But the contract cannot be changed.

Contracts and inheritance

- An **overriding function** shall have exactly the **same contract** that was declared for that function in the base class.
  - But the contract cannot be changed.

```
struct B {
public:
  virtual void f (int x) [[expects: x>0]];
  // ...
};

struct D : public B {
public:
  virtual void f (int x) override [[expects: x!=0]]; // Error
  // ...
};
```

Contracts and inheritance

- An **overriding function** shall have exactly the **same contract** that was declared for that function in the base class.
  - And it cannot be added.

Contracts and inheritance

- An **overriding function** shall have exactly the **same contract** that was declared for that function in the base class.
  - And it cannot be added.

```
struct B {
public:
  virtual void f (int x);
  // ...
};

struct D : public B {
public:
  virtual void f (int x) override [[expects: x>0]]; // Error.
  // ...
};
```

Precondition weakening

- Precondition weakening is not supported.
  - But can be simulated.

Precondition weakening

- Precondition weakening is not supported.
  - But can be simulated.

```
class A {
pubic:
  // ...
  virtual void f(int x)
    [[expects: x>0]]
  {
    [[assert: x<max]];
    // ..
  }
};
```

```
class B : public A {
pubic:
  // ...
  virtual void f(int x) override
    [[expects: x>0]]
  {
    // ...
  }
};
```

Postcondition strengthening

- Postcondition strengthening is not supported.
  - but can be simulated.

Postcondition strengthening

- Postcondition strengthening is not supported.
  - but can be simulated.

```
class A {
pubic:
  // ...
  virtual int g()
    [[ensures r: r>=0]]
  {
    // ..
  }
};
```

```
class B : public A {
pubic:
  // ...
  virtual int g() override
    [[ensures r: r>=0]]
  {
    // ...
    [[assert: result<max]];
    return result;
  }
};
```

Summary

- **Robustness** and **correctness** are orthogonal properties.
  - Use **exceptions** for **robustness**.
  - Use **contracts** for **correctness**.

Summary

- **Robustness** and **correctness** are orthogonal properties.
  - Use **exceptions** for **robustness**.
  - Use **contracts** for **correctness**.
- **Preconditions** and **postconditions** are part of the function's interface.
  - **Assertions** are part of funciont implementation.

Summary

- **Robustness** and **correctness** are orthogonal properties.
  - Use **exceptions** for **robustness**.
  - Use **contracts** for **correctness**.
- **Preconditions** and **postconditions** are part of the function's interface.
  - **Assertions** are part of funciont implementation.
- The **assertion level** specifies under which **build levels** the checking should happen.

Summary

- **Robustness** and **correctness** are orthogonal properties.
  - Use **exceptions** for **robustness**.
  - Use **contracts** for **correctness**.
- **Preconditions** and **postconditions** are part of the function's interface.
  - **Assertions** are part of funciont implementation.
- The **assertion level** specifies under which **build levels** the checking should happen.
- Start today expressing your contracts using the GSL support library.
  - `https://github.com/Microsoft/GSL`.

Work in progress implementation

- Prototype with initial implementation:
  - Supports **expects**, **ensures**, **assert**.
  - Does not support: templates/ensures, static member functions, contructors, destructors . . .
  - Aborts on contract violation.

- Maintainer: **Javier LOPEZ-GOMEZ**.

https://github.com/arcosuc3m/clang-contracts

Contracts programming for C++20

Current proposal status

J. Daniel Garcia

ARCOS Group
University Carlos III of Madrid
Spain

February 27, 2018