P3502R0
Slides for D2900R11 – Contracts for C++

Joshua Berne
Timur Doumler
Andrzej Krzemieński

ISO C++ Meeting
Wrocław, 18 November 2024

Overview

- What happened since St. Louis
- What we want to get done in EWG this week
- Why we need Contracts in C++26

What happened since St. Louis

- P2900R8
  - Added pre/post on virtual functions (as approved by EWG)
  - Made contract assertions observable checkpoints (as approved by EWG)

What happened since St. Louis

- P2900R8
  - Added pre/post on virtual functions (as approved by EWG)
  - Made contract assertions observable checkpoints (as approved by EWG)
- P2900R9
  - Constification inside contract predicates now applies to all variables, not just those with automatic storage duration

```
int g = 0;
struct X { bool m(); };

void f(int i, int* p, int& r, X x, X* px)
  pre (++g)     // error: modifying const lvalue
  pre (++i)     // error: modifying const lvalue
  pre (++(*p))  // OK
  pre (++r)     // error: modifying const lvalue
  pre (x.m())   // error: calling non-`const` member function
  pre (px->m()) // OK
  pre ([&i]{
    ++g;        // error: modifying const lvalue
    ++i;        // error: modifying const lvalue
    auto l = []{
      int j;
      ++g;      // error: modifying const lvalue
      ++j;      // OK
    };
    return i == g;
  }());
```

```
int g = 0;
struct X { bool m(); };

void f(int i, int* p, int& r, X x, X* px)
  pre (++g)      // error: modifying const lvalue
  pre (++i)      // error: modifying const lvalue
  pre (++(*p))   // OK
  pre (++r)      // error: modifying const lvalue
  pre (x.m())    // error: calling non-`const` member function
  pre (px->m()) // OK
  pre ([&i]{
    ++g;         // error: modifying const lvalue
    ++i;         // error: modifying const lvalue
    auto l = []{
      int j;
      ++g;       // error: modifying const lvalue
      ++j;       // OK
    };
    return i == g;
  }());
```

New in P2900R9

```
int g = 0;
struct X { bool m(); };

void f(int i, int* p, int& r, X x, X* px)
  pre (++g)     // error: modifying const lvalue
  pre (++i)     // error: modifying const lvalue
  pre (++(*p))  // OK
  pre (++r)     // error: modifying const lvalue
  pre (x.m())   // error: calling non-`const` member function
  pre (px->m()) // OK
  pre ([&i]{
    ++g;        // error: modifying const lvalue
    ++i;        // error: modifying const lvalue
    auto l = []{
      int j;
      ++g;      // error: modifying const lvalue
      ++j;      // OK
    };
    return i == g;
  }());
```

```
int g = 0;
struct X { bool m(); };

void f(int i, int* p, int& r, X x, X* px)
  pre (++g)     // error: modifying const lvalue
  pre (++i)     // error: modifying const lvalue
  pre (++(*p))  // OK
  pre (++r)     // error: modifying const lvalue
  pre (x.m())   // error: calling non-`const` member function
  pre (px->m()) // OK
  pre ([&i]{
    ++g;        // error: modifying const lvalue
    ++i;        // error: modifying const lvalue
    auto l = []{
      int j;
      ++g;      // error: modifying const lvalue
      ++j;      // OK
    };
    return i == g;
  }());
```

What happened since St. Louis

- P2900R8
  - Added pre/post on virtual functions (as approved by EWG)
  - Made contract assertions observable checkpoints (as approved by EWG)
- P2900R9
  - Constification inside contract predicates now applies to all variables, not just those with automatic storage duration (→ P3261R1)

What happened since St. Louis

- P2900R9
  - Constification inside contract predicates now applies to all variables, not just those with automatic storage duration (→ P3261R1)
- P2900R10
  - Allowed pre and post on coroutines (→ P2957R2)

What happened since St. Louis

- P2900R9
  - Constification inside contract predicates now applies to all variables, not just those with automatic storage duration (→ P3261R1)
- P2900R10
  - Allowed pre and post on coroutines (→ P2957R2)

presentation later today!

What happened since St. Louis

- P2900R9
  - Constification inside contract predicates now applies to all variables, not just those with automatic storage duration (→ P3261R1)
- P2900R10
  - Allowed pre and post on coroutines (→ P2957R2)
- D2900R11
  - Specified behaviour for objects passed via registers (→ P3467R0)
  - Non-reference parameters odr-used in post:
    - must be declared const on all overriding functions (→ P3464R0)
    - must be declared const even if the type is dependent (→ P3469R0)

What happened since St. Louis

- D2900R11
  - Specified behaviour for objects passed via registers (→ P3467R0)
  - Non-reference parameters odr-used in post:
    - must be declared const on all overriding functions (→ P3464R0)
    - must be declared const even if the type is dependent (→ P3469R0)
  - Added missing library bits (→ P3227R0)

What happened since St. Louis

- D2900R11
  - Specified behaviour for objects passed via registers (→ P3467R0)
  - Non-reference parameters odr-used in post:
    - must be declared const on all overriding functions (→ P3464R0)
    - must be declared const even if the type is dependent (→ P3469R0)
  - Added missing library bits (→ P3227R0)

[class.temporary] / 3

When an object of class type `X` is passed to or returned from a function, if `X` has at least one eligible copy or move constructor ([special]), each such constructor is trivial, and the destructor of `X` is either trivial or deleted, implementations are permitted to create a temporary object to hold the function parameter or result object. The temporary object is constructed from the function argument or return value, respectively, and the function's parameter or return object is initialised as if by using the eligible trivial constructor to copy the temporary (even if that constructor is inaccessible or would not be selected by overload resolution to perform a copy or move of the object).

```
class X { /* ... */ };

X f(const X* ptr)
post(r: &r == ptr) {  // guaranteed to pass only if X cannot be returned via registers
  return X{};
}

int main() {
  X x = f(&x);
}
```

```
class X { /* ... */ };

void f(X x) pre (x.g()) {  // this `x` might be at a different address...
  x.g();                   // ...than this `x` if passed via registers
}
```

Caller and callee

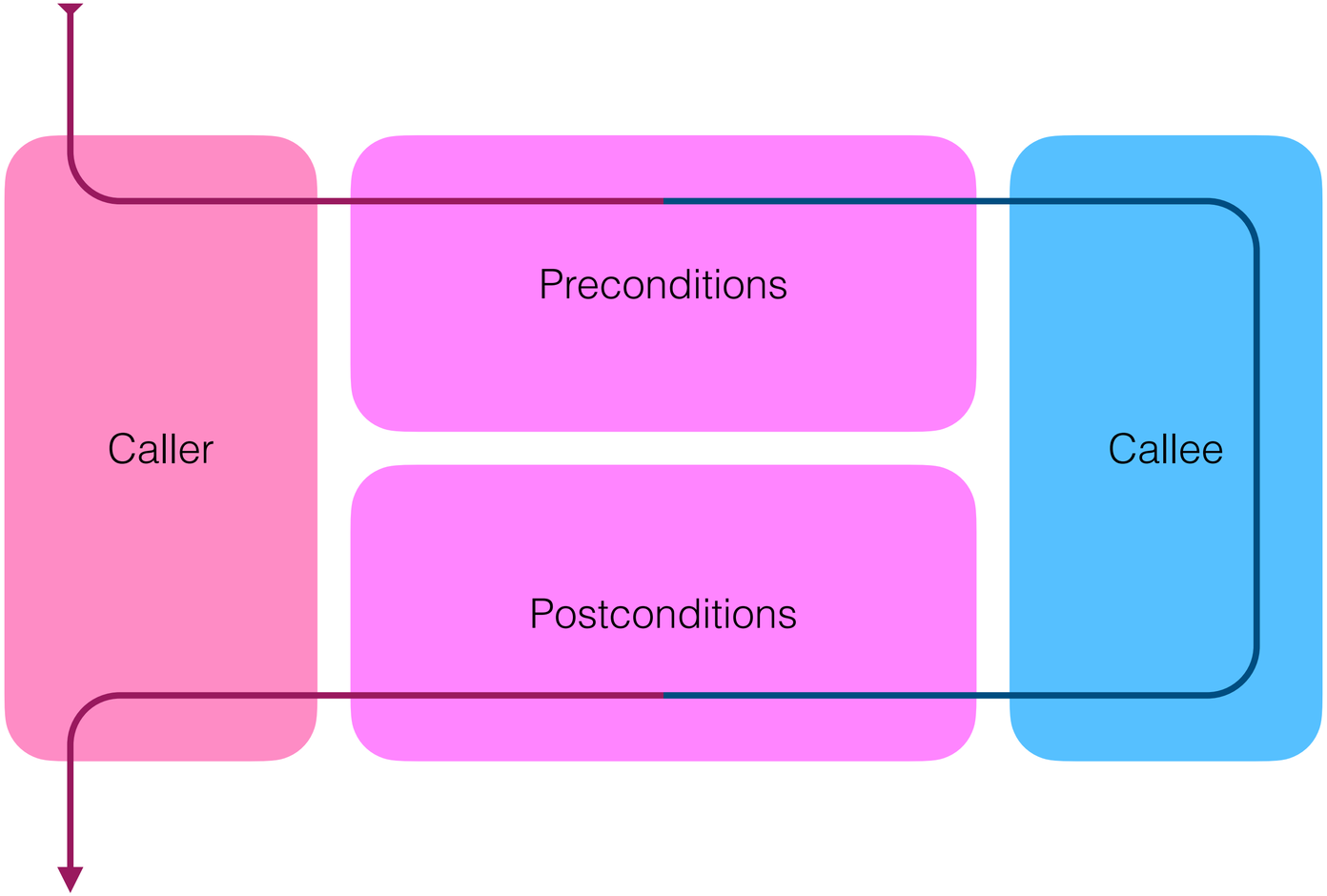


Diagram by Lisa Lippincott

Caller and callee

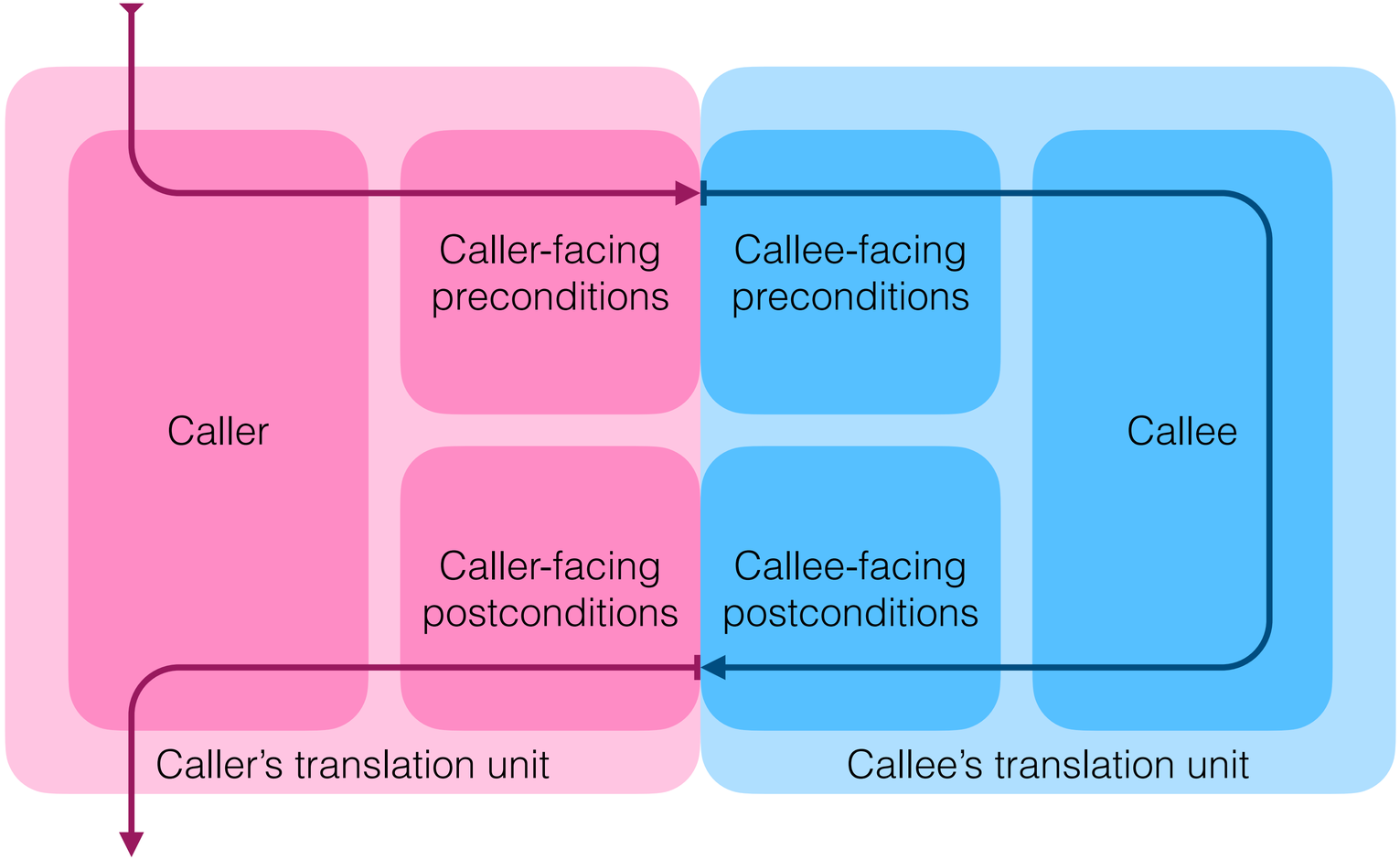


Diagram by Lisa Lippincott

```
class X { /* ... */ };

void f(X x) pre (x.g()) {  // this `x` might be at a different address...
  x.g();                   // ...than this `x` if passed via registers
}
```

New in P2900R9

```
class X { /* ... */ };
X* ptr;

void f(const X x)
post (ptr == &x) {  // guaranteed to pass only if X cannot be passed via registers
  ptr = &x;
}
```

```
class X { /* ... */ };
X* ptr;

void f(const X x)
post (ptr == &x) {  // guaranteed to pass only if X cannot be passed via registers
  ptr = &x;
}
```

New in P2900R9

What happened since St. Louis

- D2900R11
  - Specified behaviour for objects passed via registers (→ P3467R0)
  - Non-reference parameters odr-used in post:
    - must be declared const on all overriding functions (→ P3464R0)
    - must be declared const even if the type is dependent (→ P3469R0)
  - Added missing library bits (→ P3227R0)

```
int f(const int i)   // parameter odr-used in post must be const on all declarations!
post (r: r >= i);
```

```
int f(int i)
post (r: r >= i) {
  i = 0;
  return 0;
}
```

```
int f(int i)
post (r: r >= i) {
  i = 0;
  return 0;
}

void test() {
  int j = f(3);  // no contract violation detected
  // postcondition does not hold — j is now 0, which is smaller than 3 :(
}
```

```
int f(int i)
post (r: r >= i) {
  i = 0;
  return 0;
}

void test() {
  int j = f(3);  // no contract violation detected
  // postcondition does not hold — j is now 0, which is smaller than 3 :(
}

std::string g(std::string p)
post (r: starts_with(p)) {  // observes moved-from value :(
  return p;
}
```

```
struct Base {
  virtual int f(const int i) post (r: r >= i);
};

struct Derived : Base {
  int f(int i) override {
    i = 0;
    return i;
  }
};
```

```
struct Base {
  virtual int f(const int i) post (r: r >= i);
};

struct Derived : Base {
  int f(int i) override {
    i = 0;
    return i;
  }
};

void test(Base& b) {
  Integer j = b.f(3);  // checks contract of Base but no violation detected
  // precondition does not hold — j is now 0, which is smaller than 3 :(
}

int main() {
  Derived d;
  test(d);
}
```

```
struct Base {
  virtual int f(const int i) post (r: r >= i);
};

struct Derived : Base {
  int f(const int i) override {   // must be const on all declarations of all overrides
    i = 0;
    return i;
  }
};
```

New in P2900R9

What happened since St. Louis

- D2900R11
  - Specified behaviour for objects passed via registers (→ P3467R0)
  - Non-reference parameters odr-used in post:
    - must be declared const on all overriding functions (→ P3464R0)
    - must be declared const even if the type is dependent (→ P3469R0)
  - Added missing library bits (→ P3227R0)

```
template <typename T>
void f(T t) post(t > 0);

int main() {
  f<int>(1);        // error: parameter odr-used in post not declared const
}
```

```
template <typename T>
void f(T t) post(t > 0);

int main() {
  f<int>(1);        // error: parameter odr-used in post not declared const
  f<const int>(1);  // OK???
}
```

```
template <typename T>
void f(T t) post(t > 0); // error: parameter odr-used in post must be declared const here
```

What happened since St. Louis

- D2900R11
  - Specified behaviour for objects passed via registers (→ P3467R0)
  - Non-reference parameters odr-used in post:
    - must be declared const on all overriding functions (→ P3464R0)
    - must be declared const even if the type is dependent (→ P3469R0)
  - Added missing library bits (→ P3227R0)

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    std::exception_ptr evaluation_exception() const noexcept;
    assertion_kind kind() const noexcept;
    std::source_location location() const noexcept;
    evaluation_semantic semantic() const noexcept;
    bool is_terminating() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    std::exception_ptr evaluation_exception() const noexcept;
    assertion_kind kind() const noexcept;
    std::source_location location() const noexcept;
    evaluation_semantic semantic() const noexcept;
    bool is_terminating() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    std::exception_ptr evaluation_exception() const noexcept;
    assertion_kind kind() const noexcept;
    std::source_location location() const noexcept;
    evaluation_semantic semantic() const noexcept;
    bool is_terminating() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Standard Library API

```
enum class evaluation_semantic : unspecified {
  ignore = 1,
  observe = 2,
  enforce = 3,
  quick_enforce = 4
};
```

Standard Library API

```
enum class evaluation_semantic : unspecified {
  ignore = 1,
  observe = 2,
  enforce = 3,
  quick_enforce = 4
};
```

What happened since St. Louis

- D2900R11
  - Specified behaviour for objects passed via registers (→ P3467R0)
  - Non-reference parameters odr-used in post:
    - must be declared const on all overriding functions (→ P3464R0)
    - must be declared const even if the type is dependent (→ P3469R0)
  - Added missing library bits (→ P3227R0)

What happened since St. Louis

- Implementation and deployment experience! → P3460R0

What we'd like to get done in EWG this week

- Review implementers' report
  → P3460R0
- Review and approve pre/post on coroutines
  → P2957R2
- Resolve sustained objections over D2900R11
  - Lack of pre/post on function pointers
    → P2957R2
  - Evaluation of a contract predicate can have side effects and UB
    → P3362R0, P3376R0, P3386R0
  - Constification
    → P3261R1, P3478R0
- Approve D2900R11 and forward to CWG for C++26

Why we need Contracts in C++26

Why we need Contracts in C++26

Safety

Safety

Language safety
("no undefined behaviour")

- Memory safety
- Thread safety
- ...

Safety

Language safety
("no undefined behaviour")

- Memory safety
- Thread safety
- ...

Functional safety
("won't cause harm")

- System safety
- ...

Language safety
("no undefined behaviour")

Memory safety

Thread safety

...

Functional safety
("won't cause harm")

System safety

...

Language safety
("no undefined behaviour")

Memory safety

Thread safety

...

Functional safety
("won't cause harm")

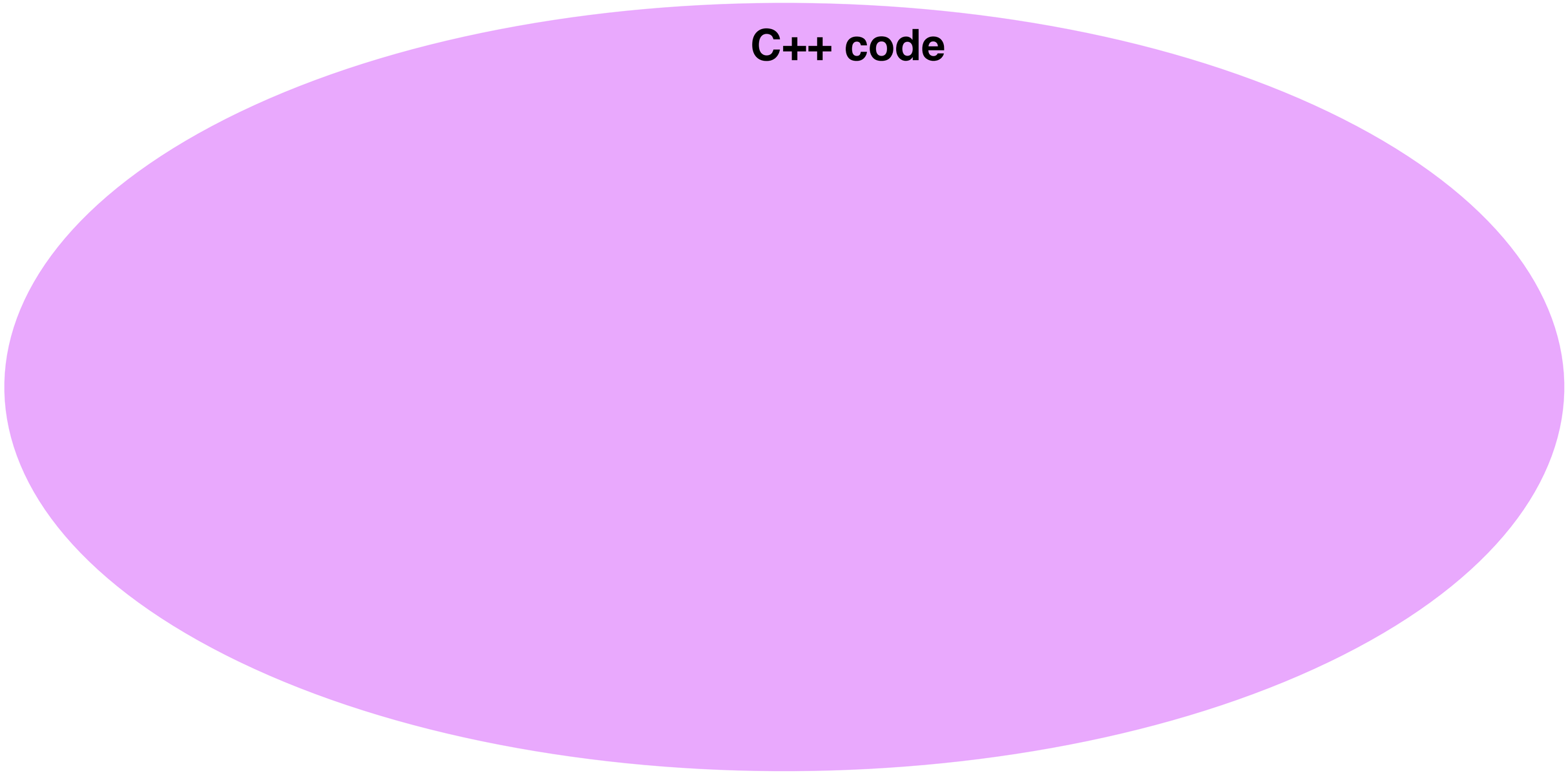
System safety

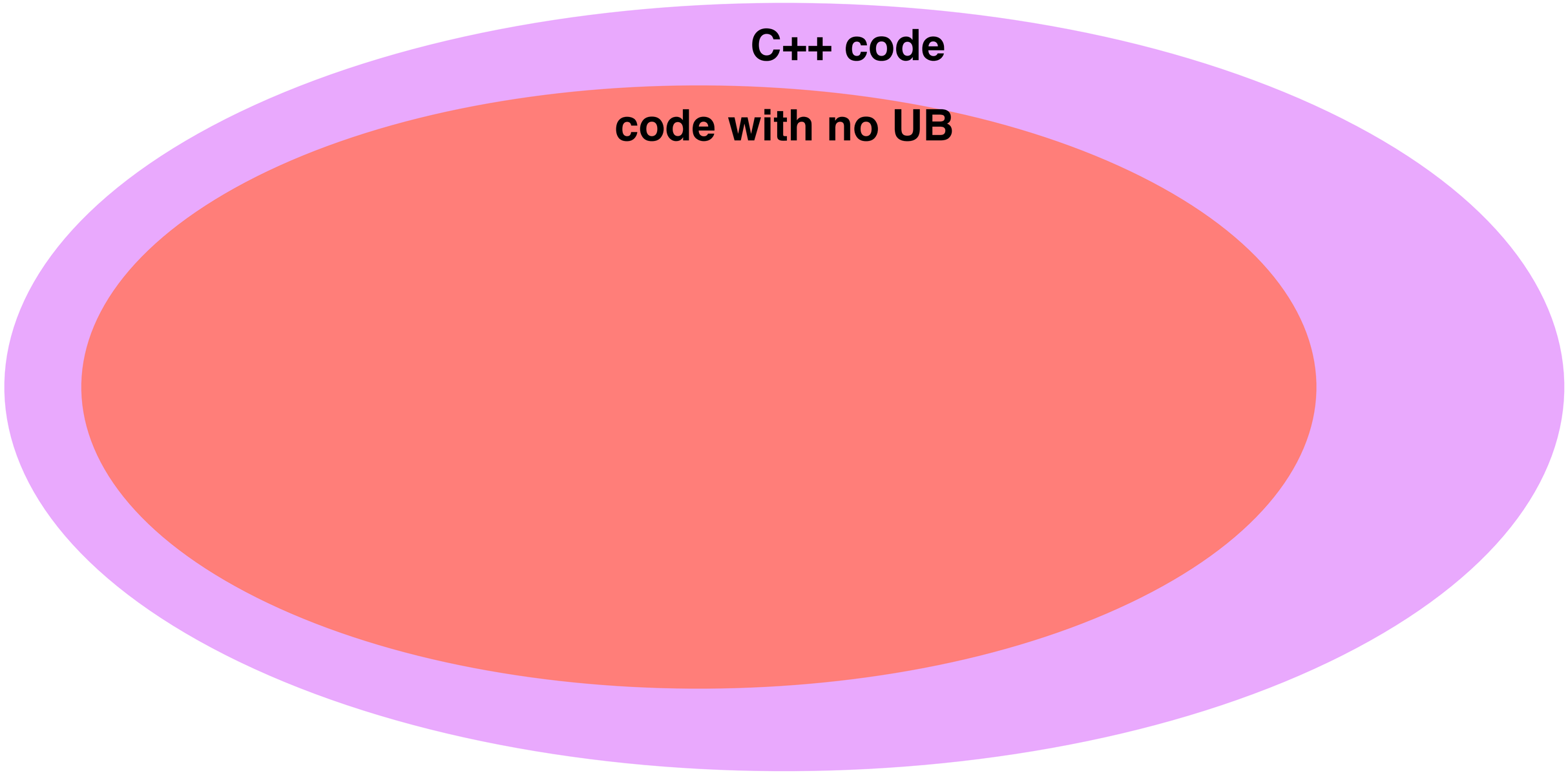
...

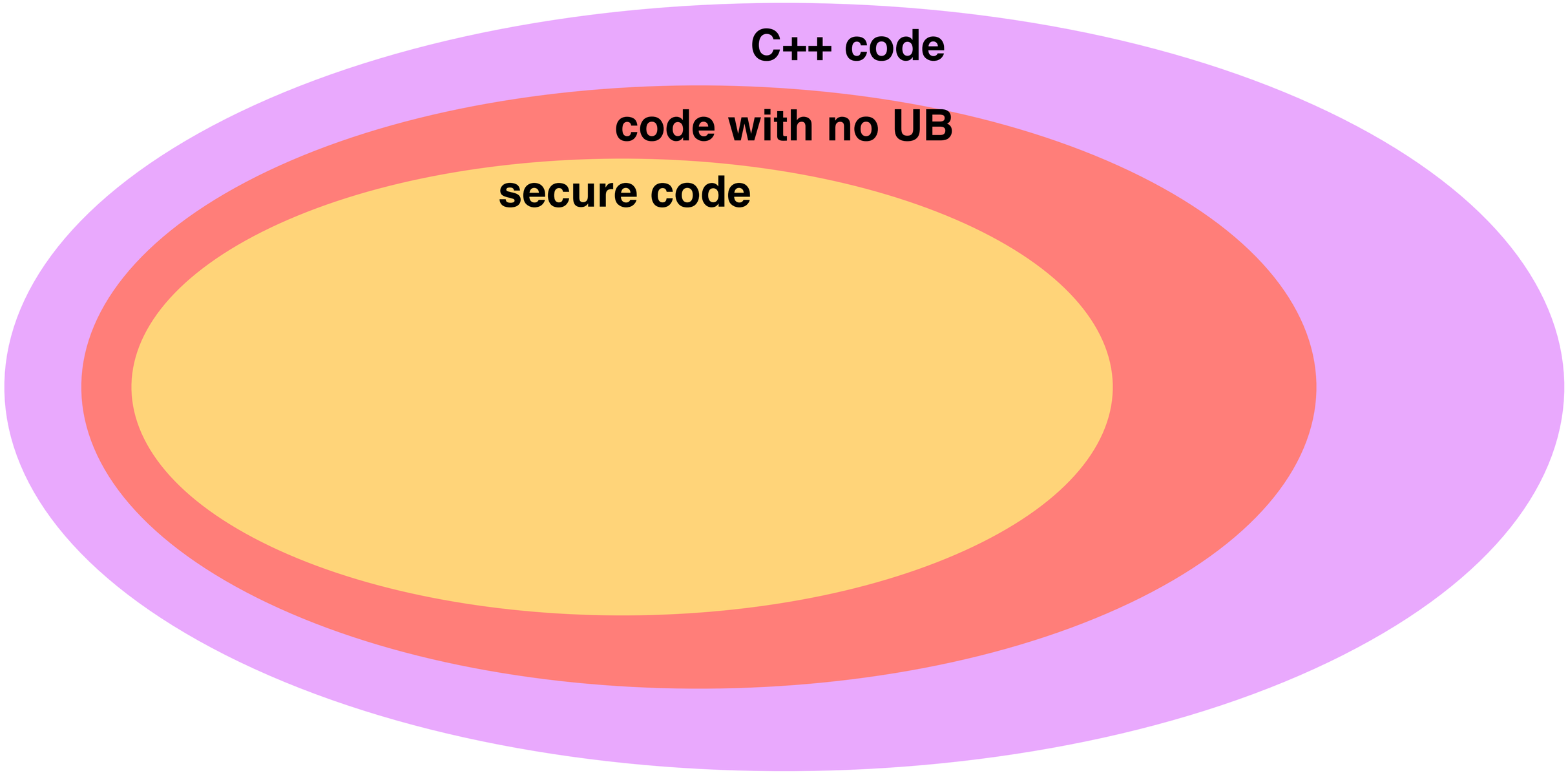
Security
("can't be exploited by malicious attacker")

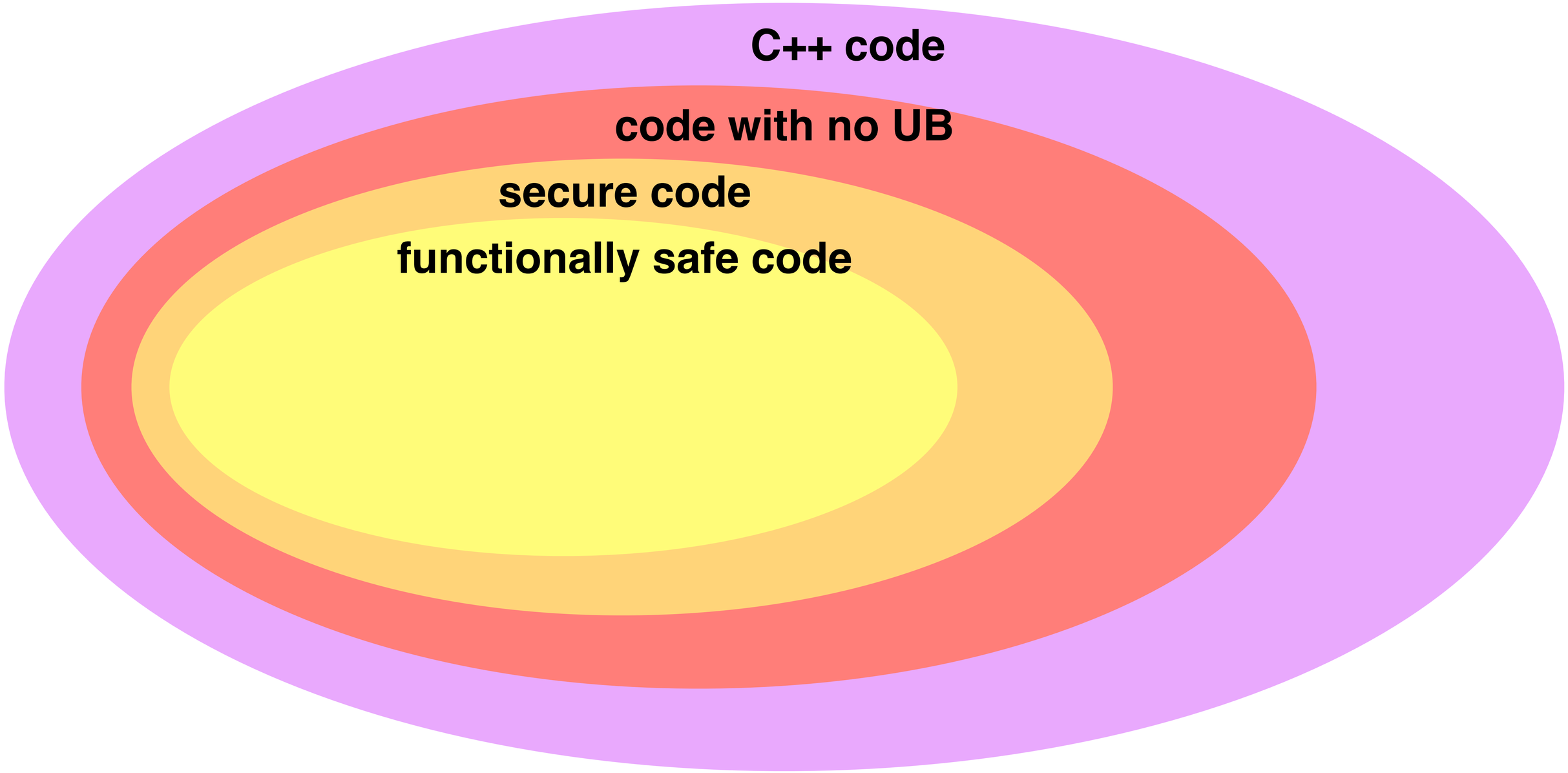
"No functional safety without security;
no security without type, resource,
& memory safety."

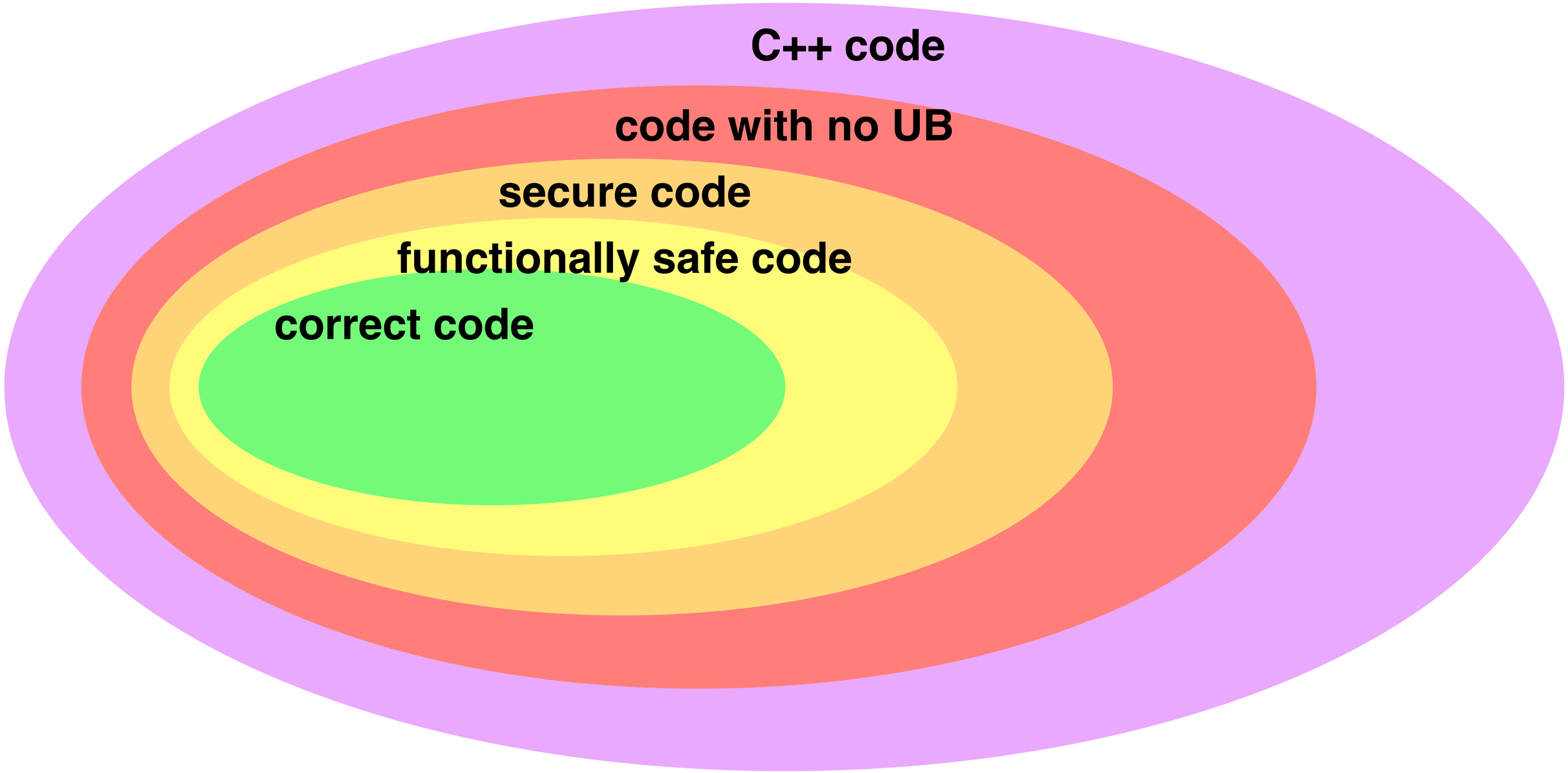
– JF Bastien

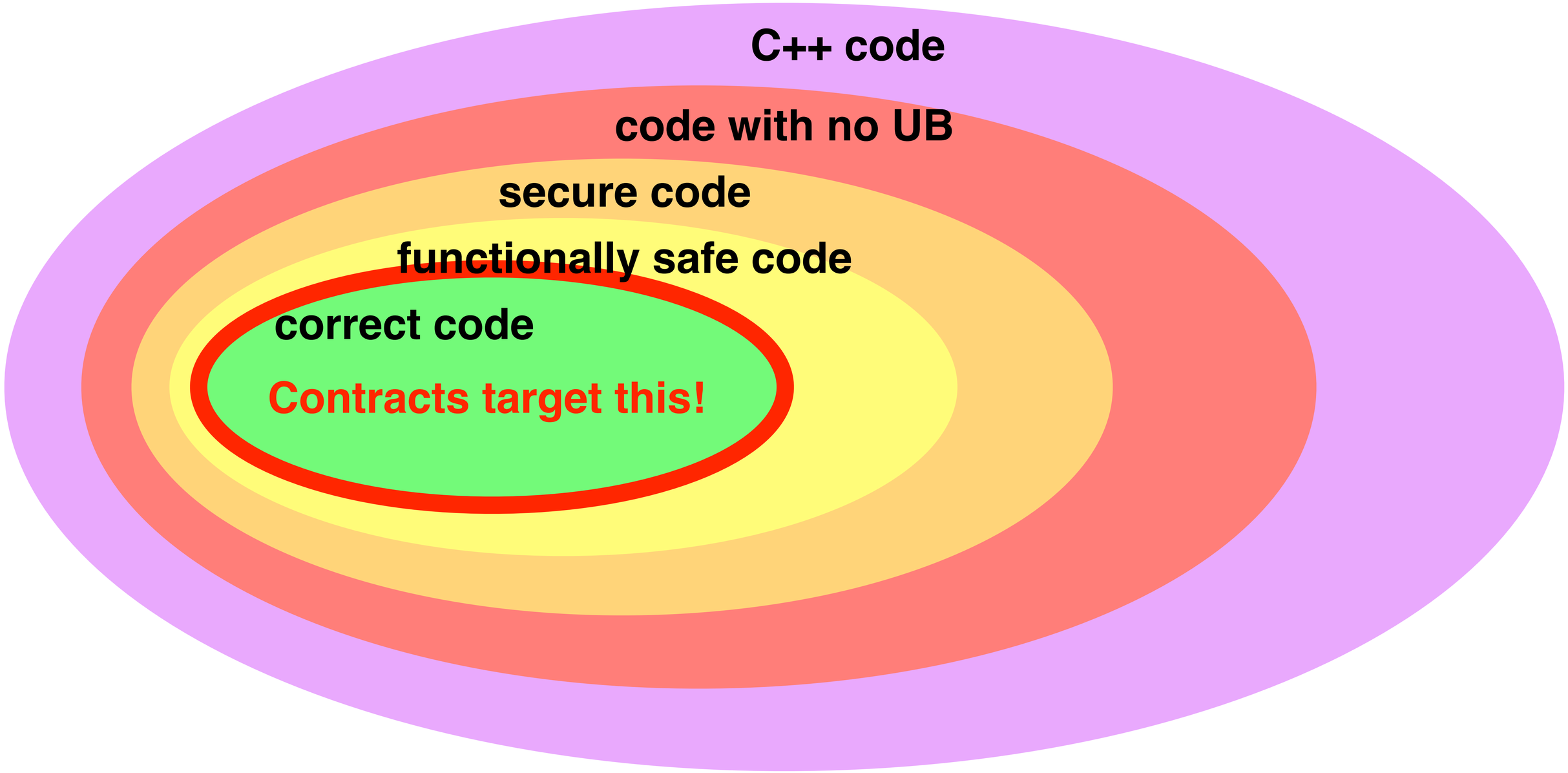

C++ code


C++ code
code with no UB


C++ code
code with no UB
secure code


C++ code
code with no UB
secure code
functionally safe code


C++ code
code with no UB
secure code
functionally safe code
correct code


C++ code
code with no UB
secure code
functionally safe code
correct code
Contracts target this!

Every additional contract assertion helps make your code more correct

```
void MyVector<T>::operator[] (size_t index)
pre (index < size()) {
  return _data[index];
}
```