# Contracts on Virtual Functions

## P3343R0 - Presentation on P3097R0

Joshua Berne - jberne4@bloomberg.net
Timur Doumler - papers@timur.audio
Gašper Ažman - gasper.azman@gmail.com
Lisa Lippincott - lisa.e.lippincott@gmail.com

2024-06-28

## Getting in the right frame of mind

# There’s always a caller’s contract.

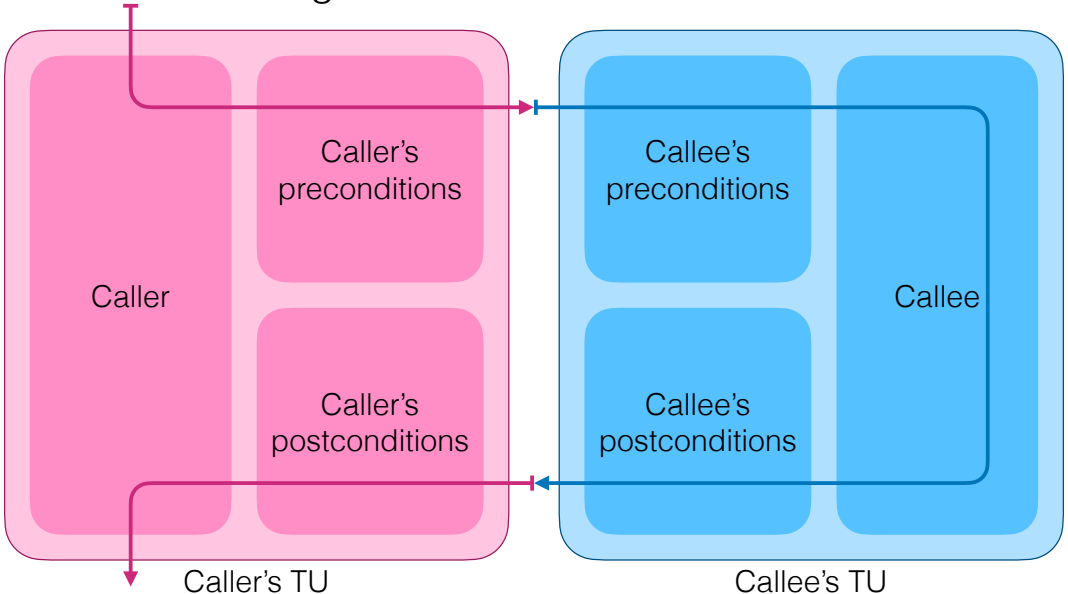
It’s in the caller’s translation unit. It may be empty.

# There’s always a callee’s contract.

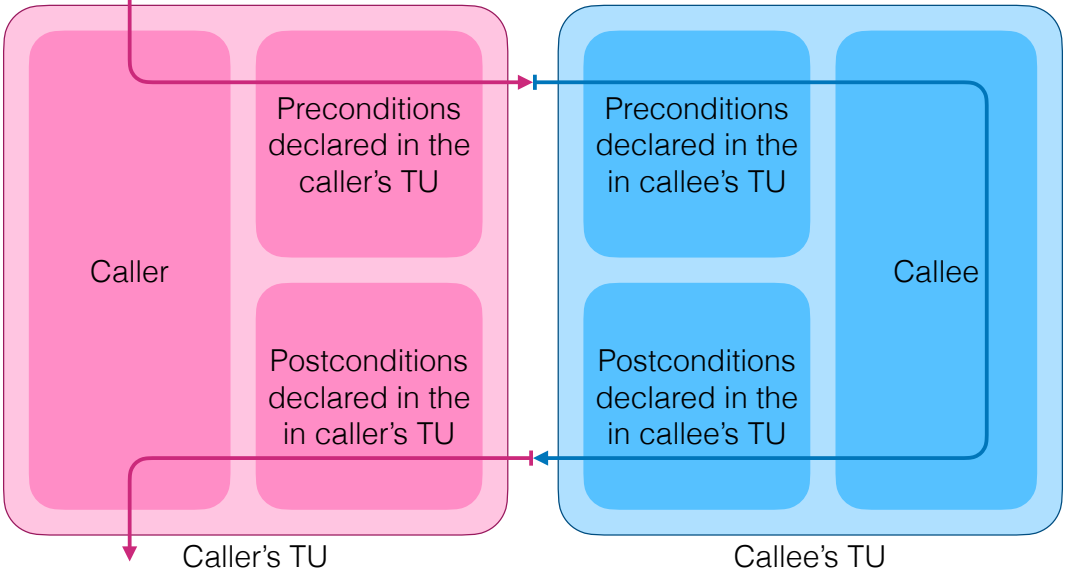
It’s in the callee’s translation unit. It may be empty.

The caller’s contract may be identical to the callee’s contract.

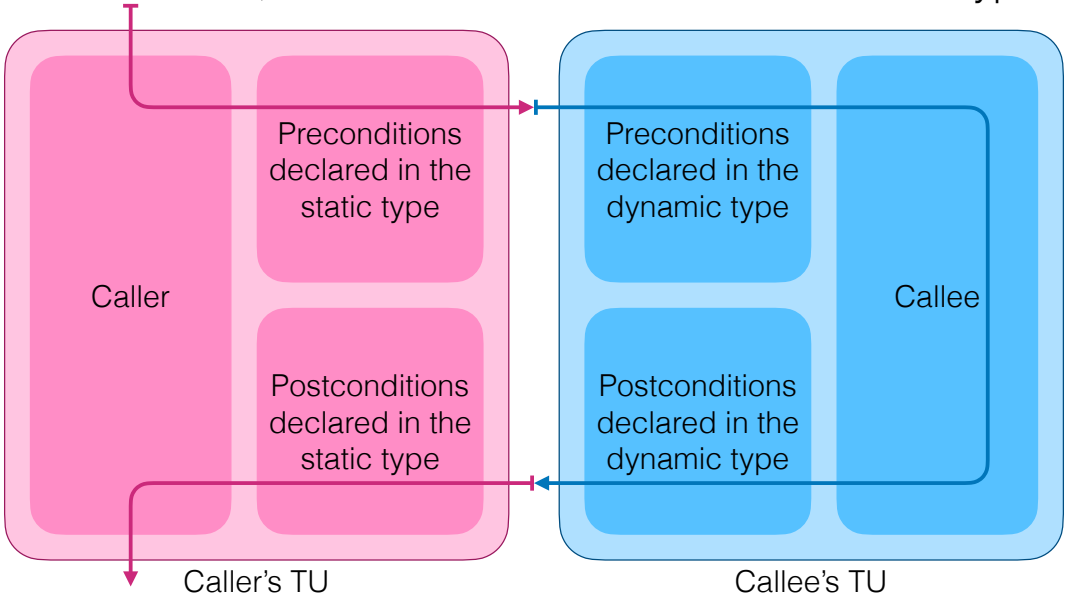
We check or ignore the assertions in both contracts.



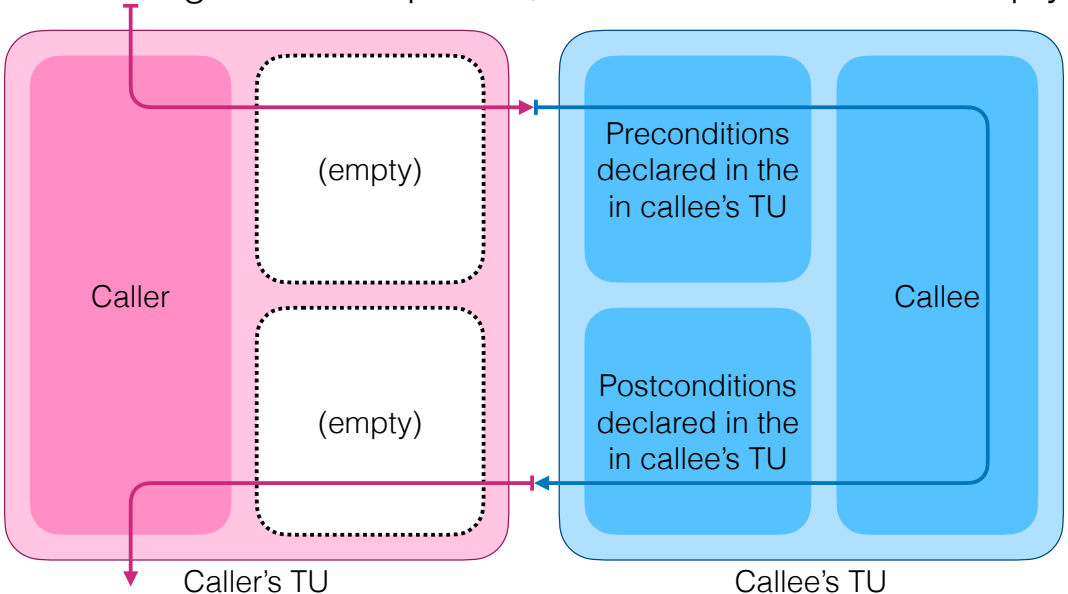
In a direct call, the two contracts are identical.



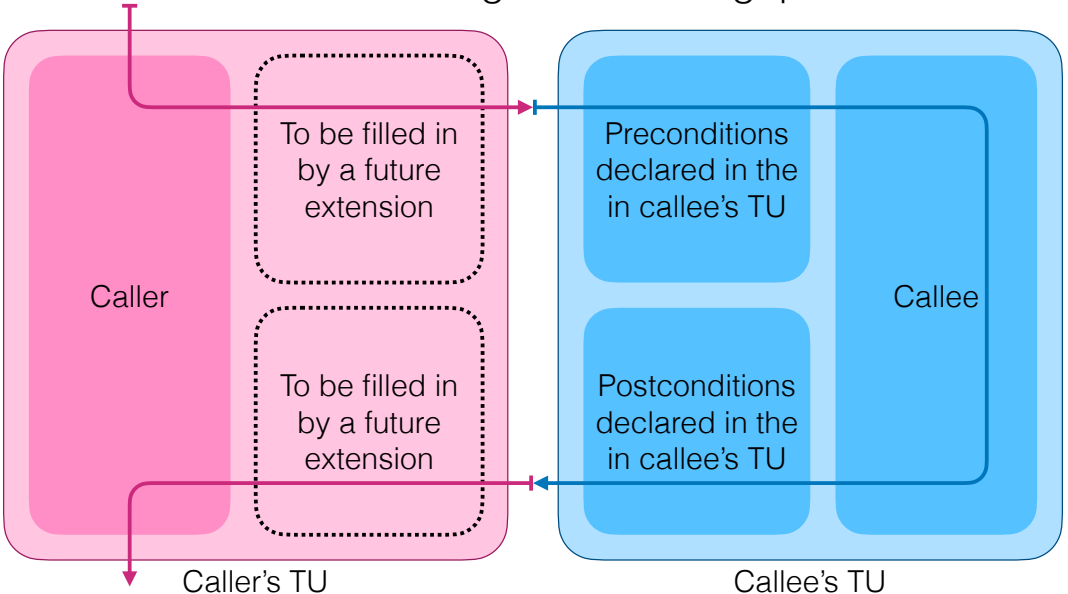
In a virtual call, the two contracts come from different types.



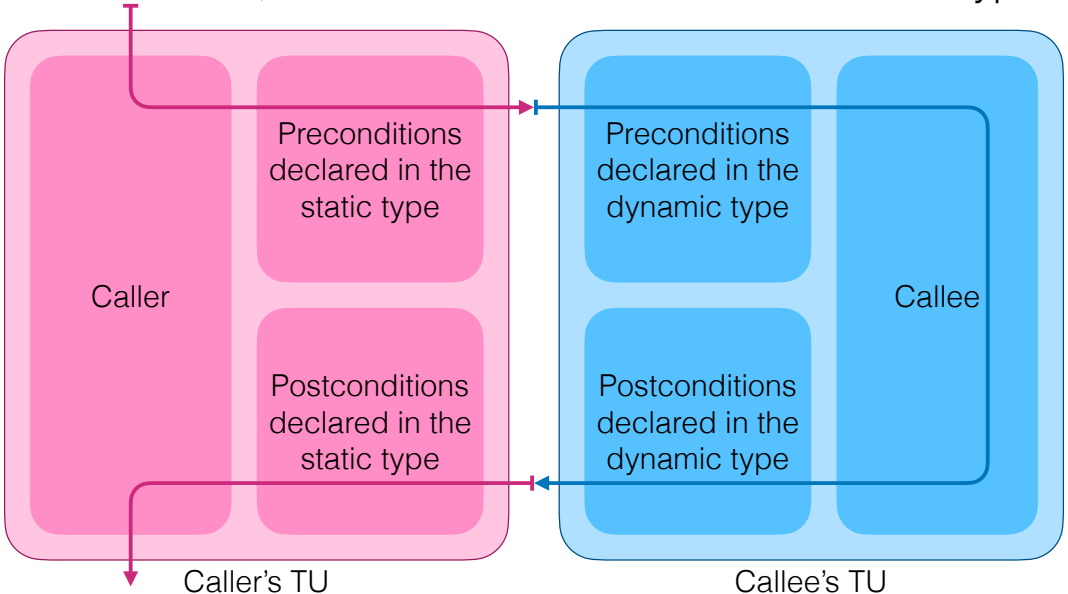
When using a function pointer, the caller's contract is empty.



We're working to fill in that gap.



In a virtual call, the two contracts come from different types.



- Lisa made that batch of slides

- Lisa made that batch of slides
- Weren't those great?

- Lisa made that batch of slides
- Weren't those great?
- I agree.

**Let's see some code**

**We need some virtual functions**

# A simple Class Hierarchy

```
struct FancyOperation {
  virtual int apply(const int x)
    pre ( x >= 0 )
    post ( r: r >= 0 )
    post ( r: r <= x ) = 0;
};
```

# A simple Class Hierarchy

```
struct FancyOperation {
  virtual int apply(const int x)
    pre ( x >= 0 )
    post ( r: r >= 0 )
    post ( r: r <= x ) = 0;
};
```

```
struct RandomDecrease : FancyOperation {
  int apply(const int x) override
    pre ( x >= 0 )
    post ( r : r >= 0 )
    post ( r : r <= x );
};
```

# A simple Class Hierarchy

```
struct FancyOperation {
  virtual int apply(const int x)
    pre ( x >= 0 )
    post ( r: r >= 0 )
    post ( r: r <= x ) = 0;
};
```

```
struct Identity : FancyOperation {
  int apply(const int x) override
    post ( r: r == x );
};
```

# A simple Class Hierarchy

```
struct FancyOperation {
  virtual int apply(const int x)
    pre ( x >= 0 )
    post ( r: r >= 0 )
    post ( r: r <= x ) = 0;
};
```

```
struct Halve : FancyOperation {
  int apply(const int x) override
    post ( r : r ==  x / 2 );
};
```

**Let's see the simplest virtual function invocation**

# Calling a Virtual Function

```
void f1()
{
  Identity identity;

  const int x = -17;



  int r = identity.apply(x);




}
```

# Calling a Virtual Function

```
void f1()
{
  Identity identity;

  const int x = -17;

  // Identity::apply no preconditions

  int r = identity.apply(x);




}
```

# Calling a Virtual Function

```
void f1()
{
  Identity identity;

  const int x = -17;

  // Identity::apply no preconditions

  int r = identity.apply(x);

  // Identity::apply postconditions
  contract_assert(r == x);    // post ( r: r == x )

}
```

## Let's use our class hierarchy

# Virtual Dispatch to a Subclass

```
void f2(FancyOperation& op)
{
  // ...
  int x = 17;
  // ...
  int r = op.apply(17);
  // ...
}
```

# Virtual Dispatch to a Subclass

```
void f2(FancyOperation& op)
{
  // ...
  int x = 17;
  // ...
  int r = op.apply(17);
  // ...
}

void test()
{
  Identity identity;
  f2(identity);
}
```

# Virtual Dispatch to a Subclass

```
void f3()
{
  Identity identity;
  FancyOperation& op = identity;

  const int x = 17;









  int r = op.apply(x);









}
```

# Virtual Dispatch to a Subclass

```
void f3()
{
  Identity identity;
  FancyOperation& op = identity;

  const int x = 17;

  // FancyOperation::apply preconditions
  contract_assert( x >= 0 );    // pre ( x >= 0 )



  int r = op.apply(x);









}
```

# Virtual Dispatch to a Subclass

```
void f3()
{
  Identity identity;
  FancyOperation& op = identity;

  const int x = 17;

  // FancyOperation::apply preconditions
  contract_assert( x >= 0 );    // pre ( x >= 0 )



  int r = op.apply(x);




  // FancyOperation::apply postconditions
  contract_assert( r >= 0 );  // post ( r : r >= 0 )
  contract_assert( r <= x );  // post ( r : r <= x )
}
```

# Virtual Dispatch to a Subclass

```
void f3()
{
  Identity identity;
  FancyOperation& op = identity;

  const int x = 17;

  // FancyOperation::apply preconditions
  contract_assert( x >= 0 );    // pre ( x >= 0 )

  // Identity::apply: no preconditions

  int r = op.apply(x);




  // FancyOperation::apply postconditions
  contract_assert( r >= 0 );  // post ( r : r >= 0 )
  contract_assert( r <= x );  // post ( r : r <= x )
}
```

# Virtual Dispatch to a Subclass

```
void f3()
{
  Identity identity;
  FancyOperation& op = identity;

  const int x = 17;

  // FancyOperation::apply preconditions
  contract_assert( x >= 0 );    // pre ( x >= 0 )

  // Identity::apply: no preconditions

  int r = op.apply(x);

  // Identity::apply postconditions
  contract_assert(r == x);   // post ( r: r == x )

  // FancyOperation::apply postconditions
  contract_assert( r >= 0 );  // post ( r : r >= 0 )
  contract_assert( r <= x );  // post ( r : r <= x )
}
```

**How about calling a base class implementation?**

# Growing the class Hierarchy

```
struct Halve : FancyOperation {
  int apply(const int x) override
    post ( r : r ==  x / 2 );
};
```

# Growing the class Hierarchy

```
struct Halve : FancyOperation {
  int apply(const int x) override
    post ( r : r ==  x / 2 );
};
```

```
struct LoggingHalve : Halve
{
  int apply(const int x) override
    post ( r : r ==  x / 2 );
};
```

# Invoking by qualified id

```
int LoggingHalve::apply(const int x)
{
  printf("Halving!");



  int r = Halve::apply(x);




  return r;
}
```

# Invoking by qualified id

```
int LoggingHalve::apply(const int x)
{
  printf("Halving!");

  // Halve::apply: no preconditions

  int r = Halve::apply(x);




  return r;
}
```

# Invoking by qualified id

```
int LoggingHalve::apply(const int x)
{
  printf("Halving!");

  // Halve::apply: no preconditions

  int r = Halve::apply(x);

  // Halve::apply Postconditions
  contract_assert( r == x / 2 );  // post ( r : r ==  x / 2 );

  return r;
}
```

**What about a member function pointer?**

# Member Function Pointers

```
void f4()
{
  Halve halve;
  FancyOperation& op = halve;
  int (FancyOperation::*apply_p)(int) = &FancyOperation::apply;

  const int x = 34;









  int r = (op.*apply_p)(x);









}
```

# Member Function Pointers

```
void f4()
{
  Halve halve;
  FancyOperation& op = halve;
  int (FancyOperation::*apply_p)(int) = &FancyOperation::apply;

  const int x = 34;

  // apply_p: empty preconditions



  int r = (op.*apply_p)(x);









}
```

# Member Function Pointers

```
void f4()
{
  Halve halve;
  FancyOperation& op = halve;
  int (FancyOperation::*apply_p)(int) = &FancyOperation::apply;

  const int x = 34;

  // apply_p: empty preconditions



  int r = (op.*apply_p)(x);




  // apply_p: empty postconditions
}
```

# Member Function Pointers

```
void f4()
{
  Halve halve;
  FancyOperation& op = halve;
  int (FancyOperation::*apply_p)(int) = &FancyOperation::apply;

  const int x = 34;

  // apply_p: empty preconditions

  // Halve::apply no preconditions

  int r = (op.*apply_p)(x);




  // apply_p: empty postconditions
}
```

# Member Function Pointers

```
void f4()
{
  Halve halve;
  FancyOperation& op = halve;
  int (FancyOperation::*apply_p)(int) = &FancyOperation::apply;

  const int x = 34;

  // apply_p: empty preconditions

  // Halve::apply no preconditions

  int r = (op.*apply_p)(x);

  // Halve::apply Postconditions
  contract_assert( r == x / 2 );  // post ( r : r ==  x / 2 );

  // apply_p: empty postconditions
}
```

**How about a deeper class hierarchy?**

# Extending Our Class Hierarchy

```
struct BoringOperation
{
  virtual int apply(const int x);
    pre ( x > 0 )
    post ( r : r >= 0 && r <= x * 2 ) = 0;
};
```

# Extending Our Class Hierarchy

```
struct BoringOperation
{
  virtual int apply(const int x);
    pre ( x > 0 )
    post ( r : r >= 0 && r <= x * 2 ) = 0;
};
```

```
struct FancyOperation : BoringOperation
{
  virtual int apply(const int x)
    pre ( x >= 0 )
    post ( r: r >= 0 )
    post ( r: r <= x ) = 0;
};
```

# Calling through a deep hierarchy

```
void f5()
{
  Halve halve;
  FancyOperation& op = halve;
  BoringOperation& bop = op;

  const int x = 51;









  int r = bop.apply(x);









}
```

# Calling through a deep hierarchy

```
void f5()
{
  Halve halve;
  FancyOperation& op = halve;
  BoringOperation& bop = op;

  const int x = 51;

  // BoringOperation::apply preconditions
  contract_assert( x > 0 );     // pre ( x > 0 )



  int r = bop.apply(x);









}
```

# Calling through a deep hierarchy

```
void f5()
{
  Halve halve;
  FancyOperation& op = halve;
  BoringOperation& bop = op;

  const int x = 51;

  // BoringOperation::apply preconditions
  contract_assert( x > 0 );     // pre ( x > 0 )



  int r = bop.apply(x);




  // BoringOperation::apply postconditions
  contract_assert( r >= 0 & r <= x * 2 ); // post ( r : r >= 0 && r <= x * 2 );
}
```

# Calling through a deep hierarchy

```
void f5()
{
  Halve halve;
  FancyOperation& op = halve;
  BoringOperation& bop = op;

  const int x = 51;

  // BoringOperation::apply preconditions
  contract_assert( x > 0 );     // pre ( x > 0 )

  // Halve::apply no preconditions

  int r = bop.apply(x);




  // BoringOperation::apply postconditions
  contract_assert( r >= 0 && r <= x * 2 ); // post ( r : r >= 0 && r <= x * 2 );
}
```

# Calling through a deep hierarchy

```
void f5()
{
  Halve halve;
  FancyOperation& op = halve;
  BoringOperation& bop = op;

  const int x = 51;

  // BoringOperation::apply preconditions
  contract_assert( x > 0 );      // pre ( x > 0 )

  // Halve::apply no preconditions

  int r = bop.apply(x);

  // Halve::apply Postconditions
  contract_assert( r == x / 2 );  // post ( r : r ==  x / 2 );

  // BoringOperation::apply postconditions
  contract_assert( r >= 0 & r <= x * 2 ); // post ( r : r >= 0 && r <= x * 2 );
}
```

**How about a wider class hierarchy?**

# Extending Our Class Hierarchy

```
struct FancyOperation
{
  virtual int apply(const int x)
    pre ( x >= 0 )
    post ( r: r >= 0 )
    post ( r: r <= x ) = 0;
};
```

# Extending Our Class Hierarchy

```
struct FancyOperation
{
  virtual int apply(const int x)
    pre ( x >= 0 )
    post ( r: r >= 0 )
    post ( r: r <= x ) = 0;
};
struct AntiFancyOperation
{
  virtual int apply(const int x)
    pre ( x <= 0 )
    post ( r : r <= 0 )
    post ( r : r >= x ) = 0;
}
```

# Extending Our Class Hierarchy

```
struct FancyOperation
{
  virtual int apply(const int x)
    pre ( x >= 0 )
    post ( r: r >= 0 )
    post ( r: r <= x ) = 0;
};
struct AntiFancyOperation
{
  virtual int apply(const int x)
    pre ( x <= 0 )
    post ( r : r <= 0 )
    post ( r : r >= x ) = 0;
}
struct Identity : FancyOperation, AntiyFancyOperation
{
  int apply(const int x) override
    post ( r: r == x );
}
```

# Using one base...

```
void f5()
{
  Identity identity;
  FancyOperation& op = identity;

  const int x = 0;

  // FancyOperation::apply preconditions
  contract_assert( x >= 0 );    // pre ( x >= 0 )

  // Identity::apply: no preconditions

  int r = op.apply(x);

  // Identity::apply postconditions
  contract_assert(r == x);   // post ( r: r == x )

  // FancyOperation::apply postconditions
  contract_assert( r >= 0 );  // post ( r : r >= 0 )
  contract_assert( r <= x );  // post ( r : r <= x )
}
```

# Using the other base...

```
void f5()
{
  Identity identity;
  AntiFancyOperation& antiOp = identity;

  const int x = 0;

  // AntiFancyOperation::apply preconditions
  contract_assert( x <= 0 );    // pre ( x <= 0 )

  // Identity::apply: no preconditions

  int r = antiOp.apply(x);

  // Identity::apply postconditions
  contract_assert(r == x);   // post ( r: r == x )

  // AntiFancyOperation::apply postconditions
  contract_assert( r <= 0 );  // post ( r : r <= 0 )
  contract_assert( r >= x );  // post ( r : r >= x )
}
```

**That's everything.**

**Really.**

## Paper Examples

# Placing Contract Annotations on Virtual Functions

```
struct Car {
  virtual void drive(float speedMph)
    pre (speedMph < 100);  // don't go too fast!
};
```

# Placing Contract Annotations on Virtual Functions

```
void testCar(Car& car) {
  car.drive(90);  // OK
  car.drive(120); // bug: precondition violation
}
```

# Widening Preconditions

```
struct FastCar : Car {
  void drive(float speed_mph) override
    pre (speed_mph < 150);
};
```

# Widening Preconditions

```
int main() {
  FastCar myFastCar;
  testCar(myFastCar);
}

void testCar(Car& car) {
  car.drive(90);  // OK
  car.drive(120); // this is too fast for a Car!
}
```

# Widening Preconditions

```
void testFastCar(FastCar& myFastCar) {
  myFastCar.drive(120);  // no problem here!
}
```

# Widening Preconditions

```
int main() {
  FastCar myFastCar;
  myFastCar.FastCar::drive(120);  // no problem here, either!
}
```

# Narrowing Preconditions

```
struct ElectricCar : Car {
  void drive(float speed_mph) override
    pre (is_charged);

  void charge() { is_charged = true }
private:
  bool is_charged = false;
};
```

# Narrowing Preconditions

```
int main() {
  ElectricCar myElectricCar;
  testCar(myElectricCar);
}

void testCar(Car& car) {
  car.drive(90);  // precondition violation: forgot to charge!
}
```

# Narrowing Preconditions

```
int main() {
  ElectricCar myElectricCar;
  myElectricCar.charge();
  testCar(myElectricCar);  // everything works now!
}
```

# Narrowing Postconditions

```
template <typename T>
struct Generator
{
  virtual T next();
};
```

# Narrowing Postconditions

```
template <typename T>
struct ConstantGenerator : Generator<T>
{
  const T &getValue() const;   // access the value that will be produced
  T next() override
    post( r : r == getValue() );
};
```

# Widening Postconditions Outside Base Class Contracts

```
struct Sqrt {
  virtual Number compute(const Number& x)
    pre( x.isReal() && x.realPart() >= 0 )
    post( r : r.isReal() && r.realPart() >= 0 );
};
```

# Widening Postconditions Outside Base Class Contracts

```
struct ComplexSqrt : public Sqrt {
  Number compute(const Number& x) override
    // wide contract
    post( r : (x.isReal() && x.realPart() >= 0)
              ? (r.isReal() && r.realPart() >= 0)
              : true );
};
```

# Widening Postconditions Outside Base Class Contracts

```
Number quadroot(Number value, const Sqrt& sqrt) {
  if (!value.isReal()  value.realPart() < 0) {
    throw std::domain_error("Must pass a nonnegative real number");
  }
  Value v = sqrt.compute( {value} );  // v is a nonnegative real number
  return sqrt.compute( {v} );         // preconditions satisfied
}
```

# Widening Postconditions

```
template <typename Base>
void TestFunction : public Base {
  void setInput(const std::vector<Value>& expectedArguments);
  const std::vector<Value>& getInput() const;

  void setOutput(const Value& value);
  Value getOutput() const;

  Value compute(const std::vector<Value>& arguments) override
    pre( arguments == getInput() )
    post( r : r == getOutput() );
};
```

# Widening Postconditions

```
Value sumSqrts(const std::vector<Value>& values, const Sqrt& sqrtFunction)

void testSumSqrts() {
  // verify that sumSqrts works correctly to sum a vector of 4s:
  std::vector<Value> values = { 4, 4, 4, 4 };
  TestFunction<Sqrt> testSqrt;
  testSqrt.setInput( {4} );
  testSqrt.setOutput( {2} );
  ASSERT( 8 == sumSqrts(values, testSqrt) );
}
```

# Multiple Inheritance

```
struct Function {
  virtual Value compute(const std::vector<Value>& arguments);
};
```

# Multiple Inheritance

```
struct UnaryFunction {
  Value compute(const std::vector<Value>& arguments) override
    pre(arguments.size() == 1);
};
struct BinaryFunction {
  Value compute(const std::vector<Value>& arguments) override
    pre(arguments.size() == 2);
};
```

# Multiple Inheritance

```
struct VariadicFunction : public UnaryFunction, public BinaryFunction {
  Value compute(const std::vector<Value>& arguments) override
    /* no preconditions */;
};
```

# Multiple Inheritance

```
struct EvenComputer {
  virtual int compute(int x)
    pre(isEven(x))
    post(r : isEven(r));
};

struct OddComputer {
  virtual int compute(int x)
    pre(isOdd(x))
    post(r : isOdd(r));
};
```

# Multiple Inheritance

```
struct Identity : EvenComputer, OddComputer {
  int compute(int x) override { return x; }
}
```