A Design by Contract Library for C

Brandon Koepke

March 24, 2016

Contents

Abstract

Design by contract is a software correctness methodology. The intention is to define a formal interface for abstract data types using preconditions, postconditions, and invariants. This library can be used to ensure that certain properties are always satisfied at runtime.

1 Motivation

Design by contract is normally used both as a form of runtime contracts (to ensure that an object doesn't invalidate it's invariants) and as a form of documentation. By forcing software developers to explicitly state method contracts we can make it much easier to understand what an implementation of a method should be doing.

Testing is another method that can be used in conjunction with design by contract to ensure that certain software properties hold. The difference with design by contract is that we can ensure that *all* implementations of a method satisfy certain properties.

In the formal methods hierarchy, design by contract sits close to the bottom (between formal specification and formal development). Since the contracts specified by a design by contract system are evaluated at runtime there is a performance penalty to using them; however, they take significantly less time during development than other formal method systems (refinement types, dependent types, theorem provers, etc) at this time.

2 Contract System

2.1 Common

We start off with the common contract definition that will be used by the implementation of many of the other contracts. We take in an expression as well as a string representation of that same expression so we can pretty-print the contract violation. We also take in the file name, line number, the function name, and a format string. Within this contract we then check whether the expression is satisfied. If it

is not then we print the error, assert failure (to aid with debugging), and finally exit with failure status. The reason that we are failing hard and fast with these contract violations is because we do not know whether the program still has valid state after a contract violation. In theory, none of these contracts should ever be violated in release code.

2a ⟨*Common Contract* 2a⟩≡

```
void contract(bool expr, const char *expr_s, const char *file,
              int line, const char *func, const char *format) {
  if (!expr) {
    fprintf(stderr, format, file, line, func, expr_s);
    assert(false), exit(EXIT_FAILURE);
  }
}
```

We also define a generic contract that asserts that it's input is non-null and also returns it's argument to the caller.

2b ⟨*Non Null Contract* 2b⟩≡

```
void *contract_non_null(const void *x, const char *x_s,
                        const char *file, int line,
                        const char *func, const char *format) {
  if (x == NULL) {
    fprintf(stderr, format, file, line, func, x_s);
    assert(false), exit(EXIT_FAILURE);
  }
  return void_cast(x);
}
```

The last common contract we will define asserts equailty between two values and returns the first argument to it's caller.

2c ⟨*Equality Contract* 2c⟩≡

```
void *contract_equal(const void *a, const void *b,
                     const char *a_s, const char *b_s,
                     const char *file, int line, const char *func,
                     const char *format) {
  if (a != b) {
    fprintf(stderr, format, file, line, func, a_s, b_s);
    assert(false), exit(EXIT_FAILURE);
  }
  return void_cast(a);
}
```

2.2 Strong

Strong contracts are contracts that should be left in release code (in other words, they are part of the public API of your module).

2.2.1 Failure

We define a simple contract that will always fail when reached. This enables us to make failure cases explicit. We use *#define* here so we can get references to the point in the source code where the contract was violated.

2d ⟨*Failure Contract Declaration* 2d⟩≡

```
#define contract_fail() _contract_fail(__FILE__, __LINE__, __func__)
void _contract_fail(const char *file, int line, const char *func);
```

3a ⟨*Failure Contract Implementation* 3a⟩≡

```
void _contract_fail(const char *file, int line, const char *func) {
  fprintf(stderr, "%s:%d: %s: Fail.\n", file, line, func);
  assert(false), exit(EXIT_FAILURE);
}
```

2.2.2 Preconditions

The next strong contract is used to assert a precondition for a method. Looking at the implementation we can see that we are just forwarding the call to the common non null handler.

3b ⟨*Requires Non Null Declaration* 3b⟩≡

```
#define contract_requires_non_null(x)                                          \
  _contract_requires_non_null((x), #x, __FILE__, __LINE__, __func__)
void *_contract_requires_non_null(const void *x, const char *x_s,
                                  const char *file, int line, const char *func);
```

3c ⟨*Requires Non Null Implementation* 3c⟩≡

```
void *_contract_requires_non_null(const void *x, const char *x_s,
                                  const char *file, int line,
                                  const char *func) {
  return contract_non_null(x, x_s, file, line, func,
                           "%s:%d: %s: Requires ‘%s’ != NULL failed.\n");
}
```

It is also useful to be able to assert equality between two values.

3d ⟨*Requires Equality Declaration* 3d⟩≡

```
#define contract_requires_equal(a, b)                                          \
  _contract_requires_equal((a), (b), #a, #b, __FILE__, __LINE__, __func__)
void *_contract_requires_equal(const void *a, const void *b, const char *a_s,
                               const char *b_s, const char *file, int line,
                               const char *func);
```

3e ⟨*Requires Equality Implementation* 3e⟩≡

```
void *_contract_requires_equal(const void *a, const void *b, const char *a_s,
                               const char *b_s, const char *file, int line,
                               const char *func) {
  return contract_equal(a, b, a_s, b_s, file, line, func,
                        "%s:%d: %s: Requires ‘%s’ == ‘%s’ failed.\n");
}
```

We can also assert arbitrary expressions.

3f ⟨*Requires Expression Declaration* 3f⟩≡

```
#define contract_requires(expr)                                                \
  _contract_requires((expr), #expr, __FILE__, __LINE__, __func__)
void _contract_requires(bool expr, const char *expr_s, const char *file,
                        int line, const char *func);
```

3g ⟨*Requires Expression Implementation* 3g⟩≡

```
void _contract_requires(bool expr, const char *expr_s, const char *file,
                        int line, const char *func) {
  contract(expr, expr_s, file, line, func,
           "%s:%d: %s: Requires ‘%s’ failed.\n");
}
```

2.2.3 Postconditions

We also redefine the above precondition functions as postcondition functions. While they do the same thing functionally, by declaring them as postconditions we can try and improve code-clarity.

4a ⟨*contract-ensures-non-null-h* 4a⟩≡

4b ⟨*Ensures Non Null Declaration* 4b⟩≡

```
#define contract_ensures_non_null(x)                                          \
  _contract_ensures_non_null((x), #x, __FILE__, __LINE__, __func__)
void *_contract_ensures_non_null(const void *x, const char *x_s,
                                 const char *file, int line, const char *func);
```

4c ⟨*Ensures Non Null Implementation* 4c⟩≡

```
void *_contract_ensures_non_null(const void *x, const char *x_s,
                                 const char *file, int line, const char *func) {
  return contract_non_null(x, x_s, file, line, func,
                           "%s:%d: %s: Ensures ‘%s’ != NULL failed.\n");
}
```

4d ⟨*Ensures Equality Declaration* 4d⟩≡

```
#define contract_ensures_equal(a, b)                                          \
  _contract_ensures_equal((a), (b), #a, #b, __FILE__, __LINE__, __func__)
void *_contract_ensures_equal(const void *a, const void *b, const char *a_s,
                              const char *b_s, const char *file, int line,
                              const char *func);
```

4e ⟨*Ensures Equality Implementation* 4e⟩≡

```
void *_contract_ensures_equal(const void *a, const void *b, const char *a_s,
                              const char *b_s, const char *file, int line,
                              const char *func) {
  return contract_equal(a, b, a_s, b_s, file, line, func,
                        "%s:%d: %s: Ensures ‘%s’ == ‘%s’ failed.\n");
}
```

4f ⟨*Ensures Expression Declaration* 4f⟩≡

```
#define contract_ensures(expr)                                                \
  _contract_ensures((expr), #expr, __FILE__, __LINE__, __func__)
void _contract_ensures(bool expr, const char *expr_s, const char *file,
                       int line, const char *func);
```

4g ⟨*Ensures Expression Implementation* 4g⟩≡

```
void _contract_ensures(bool expr, const char *expr_s, const char *file,
                       int line, const char *func) {
  contract(expr, expr_s, file, line, func, "%s:%d: %s: Ensures ‘%s’ failed.\n");
}
```

2.2.4 Invariants

Finally we define the same contracts as invariants.

4h ⟨*Invariant Expression Declaration* 4h⟩≡

```
#define contract_invariant(expr)                                              \
  _contract_invariant((expr), #expr, __FILE__, __LINE__, __func__)
void _contract_invariant(bool expr, const char *expr_s, const char *file,
                         int line, const char *func);
```

5a ⟨*Invariant Expression Implementation* 5a⟩≡

```
void _contract_invariant(bool expr, const char *expr_s, const char *file,
                         int line, const char *func) {
  contract(expr, expr_s, file, line, func,
           "%s:%d: %s: Invariant ‘%s’ failed.\n");
}
```

5b ⟨*Invariant Non Null Declaration* 5b⟩≡

```
#define contract_invariant_non_null(x)                                    \
  _contract_invariant_non_null((x), #x, __FILE__, __LINE__, __func__)
void *_contract_invariant_non_null(const void *x, const char *x_s,
                                   const char *file, int line,
                                   const char *func);
```

5c ⟨*Invariant Non Null Implementation* 5c⟩≡

```
void *_contract_invariant_non_null(const void *x, const char *x_s,
                                   const char *file, int line,
                                   const char *func) {
  return contract_non_null(x, x_s, file, line, func,
                           "%s:%d: %s: Invariant ‘%s’ != NULL failed.\n");
}
```

5d ⟨*Invariant Equality Declaration* 5d⟩≡

```
#define contract_invariant_equal(a, b)                                    \
  _contract_invariant_equal((a), (b), #a, #b, __FILE__, __LINE__, __func__)
void *_contract_invariant_equal(const void *a, const void *b, const char *a_s,
                                const char *b_s, const char *file, int line,
                                const char *func);
```

5e ⟨*Invariant Equality Implementation* 5e⟩≡

```
void *_contract_invariant_equal(const void *a, const void *b, const char *a_s,
                                const char *b_s, const char *file, int line,
                                const char *func) {
  return contract_equal(a, b, a_s, b_s, file, line, func,
                        "%s:%d: %s: Invariant ‘%s’ == ‘%s’ failed.\n");
}
```

2.3 Weak

The weak contracts are implemented in the same way as the strong contracts, the only difference is that we can compile these out without impacting the API of the application.

5f ⟨*Weak Contracts* 5f⟩≡

```
#define contract_weak_requires(expr)                                      \
  _contract_requires((expr), #expr, __FILE__, __LINE__, __func__)
#define contract_weak_requires_non_null(x)                                \
  _contract_requires_non_null((x), #x, __FILE__, __LINE__, __func__)
#define contract_weak_requires_equal(a, b)                                \
  _contract_requires_equal((a), (b), #a, #b, __FILE__, __LINE__, __func__)

#define contract_weak_ensures(expr)                                       \
  _contract_ensures((expr), #expr, __FILE__, __LINE__, __func__)
#define contract_weak_ensures_non_null(x)                                 \
  _contract_ensures_non_null((x), #x, __FILE__, __LINE__, __func__)
#define contract_weak_ensures_equal(a, b)                                 \
  _contract_ensures_equal((a), (b), #a, #b, __FILE__, __LINE__, __func__)
```

2.4 Completed files.

6a ⟨*contract.h* 6a⟩≡

```
#pragma once
#include <stdbool.h>
⟨Failure Contract Declaration 2d⟩

⟨Requires Non Null Declaration 3b⟩
⟨Requires Equality Declaration 3d⟩
⟨Requires Expression Declaration 3f⟩

⟨Ensures Non Null Declaration 4b⟩
⟨Ensures Equality Declaration 4d⟩
⟨Ensures Expression Declaration 4f⟩

⟨Invariant Non Null Declaration 5b⟩
⟨Invariant Equality Declaration 5d⟩
⟨Invariant Expression Declaration 4h⟩

⟨Weak Contracts 5f⟩
```

6b ⟨*contract.c* 6b⟩≡

```
#include "contract.h"
#include "unsafe.h"

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

⟨Common Contract 2a⟩

⟨Failure Contract Implementation 3a⟩
⟨Non Null Contract 2b⟩
⟨Equality Contract 2c⟩

⟨Requires Non Null Implementation 3c⟩
⟨Requires Equality Implementation 3e⟩
⟨Requires Expression Implementation 3g⟩

⟨Ensures Non Null Implementation 4c⟩
⟨Ensures Equality Implementation 4e⟩
⟨Ensures Expression Implementation 4g⟩

⟨Invariant Non Null Implementation 5c⟩
⟨Invariant Equality Implementation 5e⟩
⟨Invariant Expression Implementation 5a⟩
```

3 Unsafe

Definition for unsafe methods, i.e. methods that may be used only with extreme caution. For the moment we only define a void cast operator. This way we can find all void casts in our code easily and quickly and also easily find void casts that have not been checked for consistency. By using this method you are signifying that you know that the void cast is safe.

7 ⟨*unsafe.h* 7⟩≡

```
#pragma once
#define void_cast(p)((void *) p)
```