# Cbject docs

# Table of Contents

# 1. Overview

Cbject makes it easier to write object oriented code in C.

## 1.1. Features

- Classes
- Objects
- Traits
- Interfaces
- Inheritance
- Polymorphism

## 1.2. Usage

*Example 1. How to add it to a project*

Include the following header file:

```
#include "Cbject.h"
```

*Example 2. How to create an object*

```
Object * object = allocInit_(Object);
printf("%d\n", hashCode_(object));
dealloc_(object);
```

## 1.3. Object model

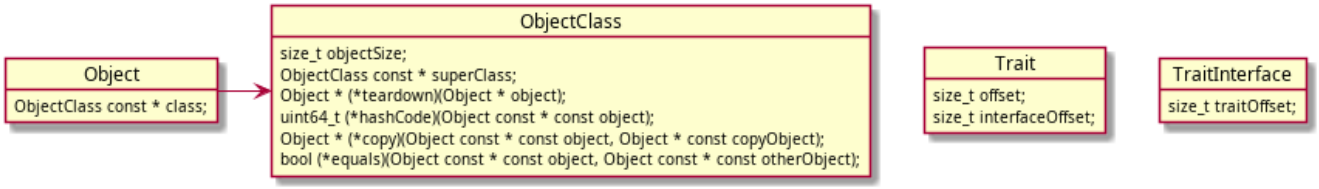


*Figure 1. Building blocks*

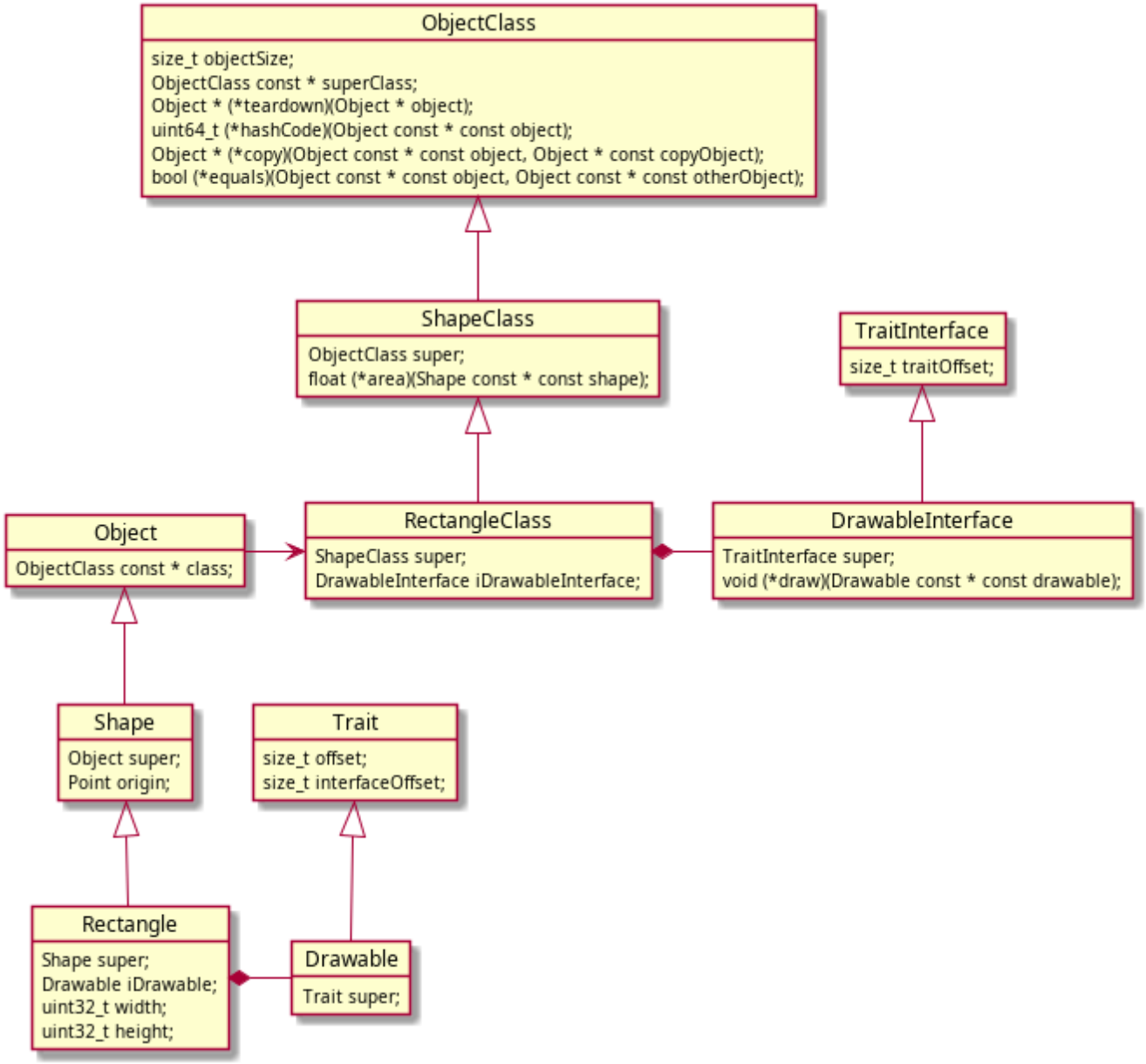


*Figure 2. Rectangle class example*

# 2. API

## 2.1. Object

### 2.1.1. Overview

The building block. All objects defined in Cbject need to extend Object.

### 2.1.2. Types

#### ObjectClass

```
typedef struct ObjectClass ObjectClass;
```

Typedef for struct ObjectClass

### Object

```
typedef struct Object Object;
```

Typedef for struct Object

### struct ObjectClass

```
struct ObjectClass {
    size_t objectSize;
    ObjectClass const * superClass;
    Object * (*teardown)(Object * object);
    uint64_t (*hashCode)(Object const * const object);
    Object * (*copy)(Object const * const object, Object * const copyObject);
    bool (*equals)(Object const * const object, Object const * const otherObject);
};
```

Definition of struct ObjectClass

*Members*

- objectSize - Size in memory of object
- superClass - Super class of object
- teardown - Function pointer for the teardown method
- hashCode - Function pointer for the hash code method
- copy - Function pointer for the copy method
- equals - Function pointer for the equals method

### struct Object

```
struct Object {
    ObjectClass const * class;
};
```

Definition of struct Object

*Members*

- class - Pointer to the class structure

### 2.1.3. Functions

#### ObjectClass_instance()

```
ObjectClass const * ObjectClass_instance(void);
```

Get ObjectClass instance

*Return*

Reference of the class instance

#### Object_alloc()

```
Object * Object_alloc(ObjectClass const * const class);
```

Allocate an object in heap memory

*Params*

- class - Class reference

*Return*

Reference of the allocated object

#### Object_dealloc()

```
Object * Object_dealloc(Object * const object);
```

Free memory allocated for an object

*Params*

- object - Object reference

*Return*

Always returns NULL

### Object_init()

```
Object * Object_init(Object * const object);
```

Initialize an object

*Params*

- object - Object reference

*Return*

Initialized object

### Object_teardown()

```
Object * Object_teardown(Object * object);
```

Teardown an object.

*Params*

- object - Object reference

*Return*

Always returns NULL

### Object_copy()

```
Object * Object_copy(Object const * const object, Object * const copyObject);
```

Make a copy of an object.

*Params*

- object - Object reference
- copyObject - Reference of a new allocated object in which to copy the original one

*Return*

Pointer to a new object (copy of the original one)

### Object_equals()

```
bool Object_equals(Object const * const object, Object const * const otherObject);
```

Compare two objects

*Params*

- object - Object reference
- otherObject - Reference for the compared object

*Return*

- true - If the objects are equal
- false - If the objects are different

### Object_hashCode()

```
uint64_t Object_hashCode(Object const * const object);
```

Get hash code of object

*Params*

- object - Object reference

*Return*

Object hash code

### Object_isOfClass()

```
bool Object_isOfClass(Object const * const object, ObjectClass const * const
class);
```

Check if an object is of a given class

*Params*

- object - Object reference
- class - Class reference

*Return*

- true - If the object is of the provided class
- false - If the object is of a different class

### 2.1.4. Macros

#### typedefClass_()

```
#define typedefClass_(className)
```

Syntactic sugar to typedef class types

*Params*

- className - Name of the class

#### class_()

```
#define class_(className)
```

Syntactic sugar to get class reference

*Params*

- className - Name of the class

*Return*

Class reference

#### initClass_()

```
#define initClass_(className)
```

Initialize a class

*Params*

- className - Name of the class

### setUpClass_()

```
#define setUpClass_(className, superClassName)
```

Class setup (initialize super, set the object size and super class)

*Params*

- className - Name of the class
- superClassName - Name of the super class

### bindClassMethod_()

```
#define bindClassMethod_(className, superClassName, methodName)
```

Bind a method of a class

*Params*

- className - Name of the class
- superClassName - Name of the super class
- methodName - Name of the method

### singleton_()

```
#define singleton_(className)
```

Syntactic sugar to get a singleton reference

*Params*

- className - Name of the class

*Return*

Singleton reference

### initSingleton_()

```
#define initSingleton_(className)
```

Initialize a singleton

*Params*

- className - Name of the class

### initObject_()

```
#define initObject_(className, ...)
```

Syntactic sugar for object initialization

*Params*

- className - Name of the class
- ...
  - object - Object reference
  - ... - Init params

*Return*

Initialized object

### sallocInit_()

```
#define sallocInit_(...)
```

Syntactic sugar to allocate and init an object in stack memory

*Params*

- ...
  - className - Name of class
  - ... - Init params

*Return*

Reference of the allocated and initialized object

### classOf_()

```
#define classOf_(object)
```

Get the class of an object

*Params*

- object - Object reference

*Return*

Class reference

### setUpObject_()

```
#define setUpObject_(className, superClassName, ...)
```

Object setup (initialize, set the object class)

*Params*

- className - Name of the class
- superClassName - Name of the super class
- ...
  - object - Object reference
  - ... - Init params

### objectSizeOf_()

```
#define objectSizeOf_(object)
```

Get the size in memory of an object

*Params*

- object - Object reference

*Return*

Object size

### traitOf_()

```
#define traitOf_(object, className, interfaceName)
```

Get trait of an object

*Params*

- object - Object reference
- className - Name of the class
- interfaceName - Name of the interface

*Return*

Trait reference

### objectMethodCall_()

```
#define objectMethodCall_(className, methodName, ...)
```

Call a method through an object

*Params*

- className - Name of the class
- methodName - Name of the method
- ...
  - object - Object reference
  - ... - Method params

*Return*

Depends on the called method

**classMethodCall_()**

```
#define classMethodCall_(className, superClassName, methodName, ...)
```

Call a method through a class

*Params*

- className - Name of the class
- superClassName - Name of the super class
- methodName - Name of the method
- ...
  - object - Object reference
  - ... - Method params

*Return*

Depends on the called method

**alloc_()**

```
#define alloc_(className)
```

Syntactic sugar to allocate an object in heap memory

*Params*

- className - Name of class

*Return*

Reference of the allocated object

### allocInit_()

```
#define allocInit_(...)
```

Syntactic sugar to allocate and init an object in heap memory

*Params*

- ...
  - className - Name of class
  - ... - Init params

*Return*

Reference of the allocated and initialized object

### dealloc_()

```
#define dealloc_(object)
```

Syntactic sugar to free memory allocated for an object

*Params*

- object - Object reference

*Return*

Always returns NULL

### teardown_()

```
#define teardown_(object)
```

Syntactic sugar to teardown an object.

*Params*

- object - Object reference

*Return*

Always returns NULL

### copy_()

```
#define copy_(className, object, copyObject)
```

Syntactic sugar to make a copy of an object.

*Params*

- className - Name of class
- object - Object reference
- copyObject - Reference of a new allocated object in which to copy the original one

*Return*

Pointer to a new object (copy of the original one)

### allocCopy_()

```
#define allocCopy_(className, object)
```

Syntactic sugar to copy object in new object allocated in heap memory

*Params*

- className - Name of class
- object - Object reference

*Return*

Reference of the allocated object (copy of the original one)

### sallocCopy_()

```
#define sallocCopy_(className, object)
```

Syntactic sugar to copy object in new object allocated in stack memory

*Params*

- className - Name of class
- object - Object reference

*Return*

Reference of the allocated object (copy of the original one)

### equals_()

```
#define equals_(object, otherObject)
```

Syntactic sugar to compare two objects

*Params*

- object - Object reference
- otherObject - Reference for the compared object

*Return*

- true - If the objects are equal
- false - If the objects are different

### hashCode_()

```
#define hashCode_(object)
```

Syntactic sugar to get hash code of object

*Params*

- object - Object reference

*Return*

Object hash code

### isOfClass_()

```
#define isOfClass_(object, className)
```

Syntactic sugar to check if an object is of a given class

*Params*

- object - Object reference
- className - Class name

*Return*

- true - If the object is of the provided class
- false - If the object is of a different class

### 2.1.5. Tests

**test_Object_class**

Test setup of ObjectClass

*Steps*

1. Get ObjectClass instance
2. Check if object size stored in class is equal to the actual object size
3. Check that the function pointers in the class are initialized

**test_Object_init**

Test initialization of Object

*Steps*

1. Allocate object on stack an initialize it
2. Check if object class points to ObjectClass instance

**test_Object_equals**

Test equals method

*Steps*

1. Allocate object on stack an initialize it
2. Check if equals method returns true when comparing object to self
3. Allocate another object on stack an initialize it
4. Check if equals method returns false when comparing the two objects

**test_Object_hashCode**

Test hashCode method

*Steps*

1. Allocate object on stack an initialize it
2. Check if hashCode method returns the address in memory of the object

**test_Object_isOfClass**

Test isOfClass method

*Preconditions*

1. Define a dummy TestClass which extends ObjectClass

*Steps*

1. Allocate object on stack an initialize it
2. Check if isOfClass method returns true when checked against Object
3. Check if isOfClass method returns false when checked against Test

**test_Object_copy**

Test copy method

*Steps*

1. Allocate object on stack an initialize it
2. Allocate another object on stack and copy the first object into it
3. Check if the memory sections occupied by the two objects are equal
4. Allocate another object on heap and copy the first object into it
5. Check if the memory sections occupied by the two objects are equal
6. Deallocate the object from the heap memory

# 2.2. Trait

## 2.2.1. Overview

TODO

## 2.2.2. Types

### TraitInterface

```
typedef struct TraitInterface TraitInterface;
```

Typedef for struct TraitInterface

### Trait

```
typedef struct Trait Trait;
```

Typedef for struct Trait

### struct TraitInterface

```
struct TraitInterface {
    size_t traitOffset;
};
```

Definition of struct TraitInterface

*Members*

- traitOffset - Offset of trait in containing object

**struct Trait**

```
struct Trait {
    size_t offset;
    size_t interfaceOffset;
};
```

Definition of struct Trait

*Members*

- offset - Offset of Trait in container Object
- interfaceOffset - Offset of TraitInterface in container ObjectClass

### 2.2.3. Functions

**TraitInterface_instance()**

```
TraitInterface const * TraitInterface_instance(void);
```

Get TraitInterface instance

*Return*

Reference of the trait interface

**Trait_init()**

```
Trait * Trait_init(Trait * const trait);
```

Initialize a trait

*Params*

- trait - Trait reference

*Return*

Initialized trait

### 2.2.4. Macros

#### typedefInterface_()

```
#define typedefInterface_(interfaceName)
```

Syntactic sugar to typedef interface types

*Params*

- interfaceName - Name of the interface

*Return*

Interface reference

#### interface_()

```
#define interface_(interfaceName)
```

Syntactic sugar to get interface reference

*Params*

- interfaceName - Name of the interface

*Return*

Interface reference

#### initInterface_()

```
#define initInterface_(interfaceName)
```

Initialize a class

*Params*

- interfaceName - Name of the interface

### setUpInterface_()

```
#define setUpInterface_(interfaceName, superInterfaceName)
```

Interface setup (initialize super)

*Params*

- interfaceName - Name of the interface
- interfaceName - Name of the super interface

### bindInterfaceMethod_()

```
#define bindInterfaceMethod_(interfaceName, superInterfaceName, methodName)
```

Bind a method of an interface

*Params*

- interfaceName - Name of the interface
- superInterfaceName - Name of the super interface
- methodName - Name of the method

### setUpInterfaceOf_()

```
#define setUpInterfaceOf_(className, interfaceName)
```

Interface setup in class (initialize super, set the trait offset in container object)

*Params*

- className - Name of the class
- interfaceName - Name of the interface

### bindInterfaceMethodOf_()

```
#define bindInterfaceMethodOf_(className, superClassName, interfaceName,
methodName)
```

Bind a method of an interface

*Params*

- className - Name of the class
- superClassName - Name of the super class
- interfaceName - Name of the interface
- methodName - Name of the method

### offsetOf_()

```
#define offsetOf_(trait)
```

Get offset of a trait in container object

*Params*

- trait - Trait reference

*Return*

Offset of trait in container object

### objectOf_()

```
#define objectOf_(trait)
```

Get container object from a trait

*Params*

- trait - Trait reference

*Return*

Reference of the container object

### interfaceOffsetOf_()

```
#define interfaceOffsetOf_(trait)
```

Get the interface offset in container class

*Params*

- trait - Trait reference

*Return*

Offset of interface in container class

### interfaceOf_()

```
#define interfaceOf_(trait)
```

Get the interface of a trait

*Params*

- trait - Trait reference

*Return*

Interface reference

### initTrait_()

```
#define initTrait_(interfaceName, ...)
```

Syntactic sugar for trait initialization

*Params*

- interfaceName - Name of the interface
- ...
  - trait - Trait reference
  - ... - Init params

*Return*

Initialized trait

### setUpTraitOf_()

```
#define setUpTraitOf_(className, interfaceName, ...)
```

Trait setup (initialize, set the trait offset and interface offset)

*Params*

- className - Name of the class
- interfaceName - Name of the interface
- ...
  - object - Object reference
  - ... - Init params

### traitMethodCall_()

```
#define traitMethodCall_(interfaceName, methodName, ...)
```

Call a method through a trait

*Params*

- interfaceName - Name of the interface
- methodName - Name of the method
- ...
  - trait - Trait reference
  - ... - Method params

*Return*

Depends on the called method

### interfaceMethodCall_()

```
#define interfaceMethodCall_(className, interfaceName, methodName, ...)
```

Call a method through an interface

*Params*

- className - Name of the class
- interfaceName - Name of the interface
- methodName - Name of the method
- ...
  - trait - Trait reference
  - ... - Method params

*Return*

Depends on the called method

# 2.3. Utils

## 2.3.1. Overview

TODO

## 2.3.2. Types

### Any

```
typedef void Any;
```

Typedef for Any

*Remark*

To be used with pointers to anything

## 2.3.3. Macros

### to_()

```
#define to_(typeName, instance)
```

Cast an instance to the provided typeName

*Params*

- typeName - Name of the type (class or interface)
- instance - Instance to cast

*Return*

Instance cast to the provided typeName

### extends_()

```
#define extends_(typeName)
```

Syntactic sugar to extend a type (adds the member super to the structure)

*Remark*

Should be used as the first member in the structure

*Params*

- typeName - Name of the type

### extends_()

```
#define implements_(typeName)
```

Syntactic sugar to compose a type with the provided typeName

*Remark*

Should be used after extends_() macro

*Params*

- typeName - Name of the type

### lengthOf_()

```
#define lengthOf_(array)
```

Get length of an array

*Params*

- array - Array for which to get the length

### lengthOf_()

```
#define salloc_(typeName)
```

Syntactic sugar to allocate memory on the stack

*Params*

- typeName - Name of type

### ignore_()

```
#define ignore_(var)
```

Syntactic sugar to ignore unused variables

*Params*

- var - Variable to be ignored

### VaArgs_first_()

```
#define VaArgs_first_(...)
```

Get first argument from *VA_ARGS*

*Params*

- ... - *VA_ARGS*

### VaArgs_rest_()

```
#define VaArgs_rest_(...)
```

Get list of arguments from *VA_ARGS* except the first

*Remark*

- Comma is added before the list
- Supports max 10 arguments

*Params*

- *...* - *VA_ARGS*