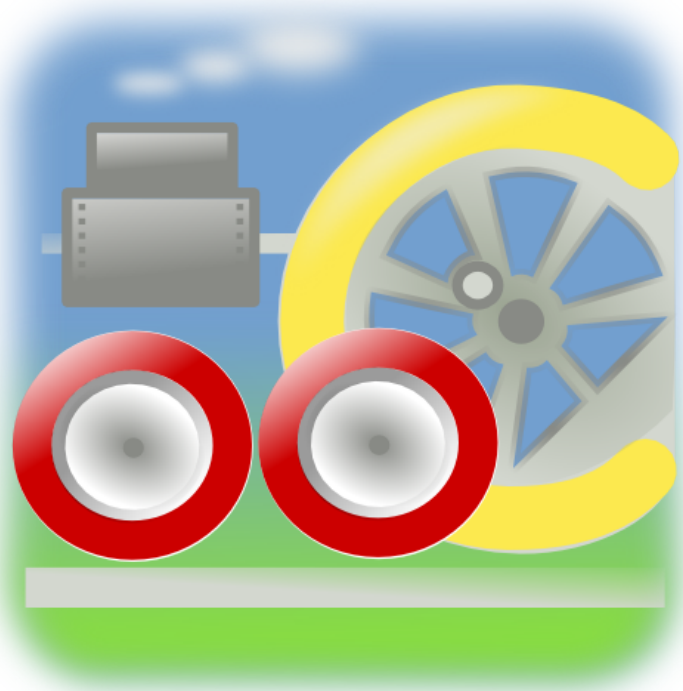
Object Oriented C (ooc) toolkit

for version 1.3a, 8 January 2012



Tibor Miseta

This manual is for Object Oriented C (ooc) toolkit (version 1.3a, 8 January 2012), which is a lightweight collection of tools for Object Oriented programming approach in ANSI-C.

Table of Contents

1 Introduction

Object Oriented C toolkit, or shortly **ooc** has been created with the intention to enable to write object oriented code easily using standard **ANSI-C**, with all the possible type checks. It is very important being ANSI-C compliant, because the main goal was the portability even for systems that lack a C++ compiler.

The other goal was keeping it lightweight, being able to port it onto small computers as well, like embedded systems. The implementation has a C++ like approach, using *classes*, *objects*, and *exceptions*. The ooc also incorporates *single inheritance* and with the help of interfaces and mixins a kind of *multiple inheritance*.

There are many similar kits out there, but I found most of them either too complicated for writing a fully controllable, really *portable* code, or inconvenient to use it for writing readable programs. So I have started from scratch.

The **ooc** toolkit comes with some container classes, a unit testing helper class and a tool that let's you generate classes from several templates easily.

In this manual I will cover only issues related to the implementation and use of **ooc**, I assume that the reader is familiar with the Object Oriented Programming and has a good knowledge about C and C++ and their most common internal implementation.

More detailed information on the use of **ooc** can be found in the ooc API documentation.

2 Objects and Classes

Class is the type of an Object instance. It specifies its data and function members and methods. We use the same terminology as in C++ here.

Before an Object could be used, it must be instantiated.

The Classes and Objects are located in the memory, as the following section describes.

2.1 Underlying data structure

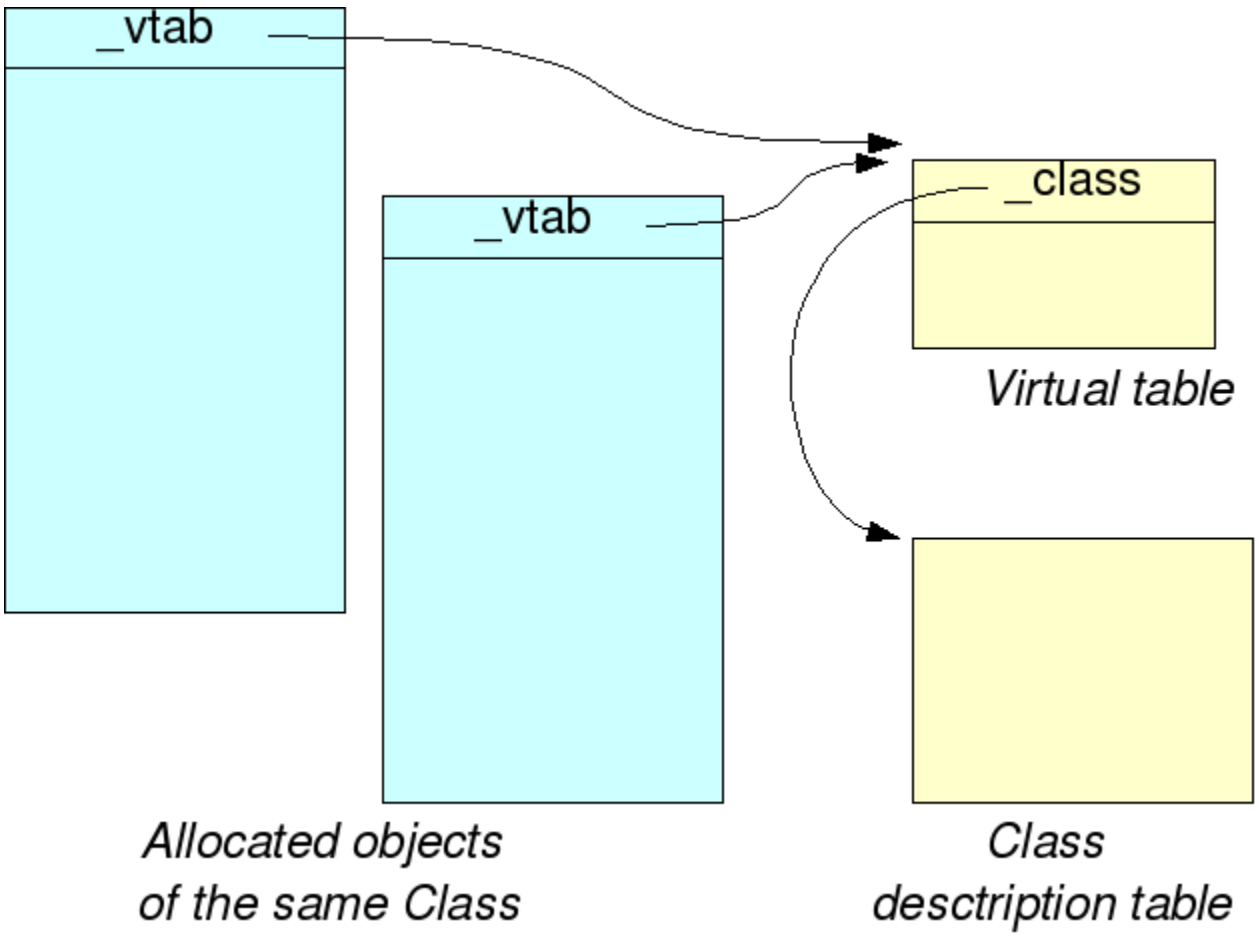


Figure 2.1: Underlying data structure

The Class description table and the Virtual table are allocated statically in compilation time while the Object instances are allocated dynamically on the heap in run time. The Class allocation table is fully initialized at compilation time, while the Virtual Table must

be initialized in run time before the Class is used. Due to this limitation it is not possible allocating Objects statically on the heap.

There is always a Virtual table, even if the class does not have a virtual function. In this case it is just a single pointer to the Class description table.

2.2 Inheritance

In ooc single inheritance is supported. The physical implementation of inheritance is embedding the parent class members into the beginning of the instantiated object.

Real multiple inheritance is not supported because of considering run time effectiveness on slower computers; plus trying to avoid complex inheritance problems that may occur in case of multiple inheritance, and a good solution for them would require more support from the compiler.

However since version 1.3 **ooc** supports the use of interfaces and mixins providing a kind of multiple inheritance.

In every class definition macros we use two parameters:

- First parameter is the name of the class, while the
- Second parameter is the name of the parent class.

If a class is a base class (has no parent class), we shall mark it as it parent class was `Base`. Therefore `Base` is a reserved class name in **ooc**!

```
DeclareClass( String, Base );       /* String is a Base class */

DeclareClass( Utf8, String );       /* Utf8 is a class inherited from String */
```

2.3 Class data members

Class definitions are basically nested struct definitions. That means that you can access data members via their names, as they were accessed as struct members. There is an important rule, that accessing the parent class's data members requires a prefix with the parent class's name before the data member name. This is because the standard ANSI C does not allow the use of unnamed struct, and I wanted to be ANSI compliant for better portability.

```
ClassMembers( String, Base )

    char *    cstr;
    int       length;

EndOfClassMembers;

ClassMembers( Utf8, String )
```

```
     int       num_of_chars;

EndOfClassMembers;

/*********************************
 *  Accessing data members
 */

    String my_string;
    Utf8   my_utf8;
    int i;

    i =  my_string->length;        /* Accessing a class member */

    i =  my_utf8->num_of_chars;    /* Accessing a class member */

    i =  my_utf8->String.length;   /* Accessing class member inherited from the parent
```

2.4 Member functions

A class member function is a normal C function, but there is a very important rule: the first function parameter of a member function is always a class instance object, and this first parameter can not be omitted.

```
void   str_upper( String ); /* Declaring a member function */

void   str_upper( String self ) /* Defining a member function */
{
    int i;
    assert( ooc_isInstanceOf( self, String ) );

    for( i=0; i<self->length; i++ )
        self->cstr[i] = cupper( self->cstr[i] );
}

str_upper( my_string );   /* Calling a member function */
```

As a naming convention it is a good idea to start all class member function's name with the name of the class, or with a meaningful abbreviation.

2.5 Virtual functions

Virtual functions have the same requirement: their mandatory first parameter is an object instance pointer.
Virtual functions are implemented as static functions in the class implementation file, and

the class's virtual table holds pointers to these static functions.
Virtual functions are called via their function pointers in the vtable.

The vtable itself is basically a struct holding function pointers to the implemented static functions. The calling via these function pointers provides us the capability for compilation time type and parameter checking.

For virtual function calls we use the macros and inline functions.
For those compilers that the inline functions are not supported there is a function version for virtual function calls, but that is slower of course. This is the price for the better type safety.

```
/* Defining a virtual function */
Virtuals( String, Base )

    int (* str_get_tokens)( String );

EndOfVirtuals;

/* Calling a virtual function; from the user point of view */
int len;
String my_string;

len = StringVirtual( my_string )->str_get_tokens( my_string );

/* Implementing the virtual function in the class implementation file */

static int virtual_str_get_tokens( String self )
{
    /* doing some important here with self */
    return result;
}

/* Initializing the virtual table in the class initialization handler */

static void String_init( )
{
    StringVtableInstance.str_get_tokens   =   virtual_str_get_token;
}
```

2.5.1 Overridden virtual functions

Overriding the parent class's virtual functions is very easy in the class implementation file. It can be done in the class initialization code.

```
/* Defining the virtual table */
```

```
Virtuals( Utf8, String )
                                    /* In this case there are no new Virtual functions,
EndOfVirtuals;

/* Implementing the virtual function in the class implementation file */

static int utf8_get_tokens( String self )
{
    /* doing some important here with self */
    return result;
}

/* Overriding the parent's virtual function pointer in the virtual table in the class

static void Utf8_init( Class class )
{
    Utf8VtableInstance.String.str_get_tokens   =   utf8_get_token;
}

/* In the class's user code you can call the virtual in the same way: */

len = Utf8Virtual( my_utf8 )->String.str_get_tokens( (String) my_utf8 );
```

2.5.2 Calling parent's virtual functions

If you would like to call the parent class's virtual function (this may be necessary in the implementation code when you would like to chain the new class's virtual function to the parent's one, you can use other macro accessing the original (non-overridden) function:

```
static int utf8_get_tokens( String self )
{
    /* doing some important stuff here with self,
       then chaining to the original parent's function: */

    return Utf8ParentVirtual( self )->str_get_tokens( self );
}
```

2.6 Class description table

The Class description table is completely hidden from the user of the class. It is a static struct in the heap, created and initialized at compilation time. The identifier of the class is the address of this class description table, so you must refer to the class with the "address of" operator and the class name.

You can allocate the class description table, the virtual table and some other helpers with a single macro:

```
AllocateClass( String, Base );

AllocateClass( Utf8, String );
```

3 Exception handling

However the exception handling from the user's point of view is very similar to the exception handling in C++, there are very significant differences that you must keep in mind! In **ooc** the key differences are:

- There is *no* stack unwinding!
- You must consider the side effects of optimizing compilers.
- There is an additional `finally` option.
- You must close the section with an `end_try` statement.
- Every executed `try` must have an executed `end_try`!

Being able to use the exceptions you must include the `exception.h` header file, and call `ooc_init_class( Exception );` at the very beginning of your code! Because **ooc** uses the exceptions internally, you must always initialize the `Exception` class in your code before using any other **ooc** features!

3.1 Throwing an exception

Throwing an exception is very easy with the `ooc_throw()` instruction. The parameter of the `ooc_throw()` is any `Object`. The object is "owned" by the exception handling mechanism and will be deleted by it, so never delete it yourself, and never throw an object, that you would like to use later. In practice I recommend throwing newly created objects of `Exception` class or its subclasses, like:

```
if( error )
    ooc_throw( str_exception_new( str_error_code ));
```

You can use `ooc_rethrow()` as well for passing the actual exception to the caller, but only inside a `catch()` or in a `catch_any` block.

3.2 Catching an exception

You can catch the thrown exceptions with the `catch()` or `catch_any` blocks.

The `catch( Class )` block catches the objects of the specified class or its parent (super-) classes. The caught object is stored in a variable called `Exception exception`, that is automatically defined and can be used only within the `catch` block.

The `catch_any` block catches all exceptions that were not handled by the earlier `catch` blocks. It is typically used for cleanup and rethrow of exception that could not be handled locally. You can use as many `cath()` blocks, as many you need, plus one `catch_any` block as the last one. Be careful with the ordering of the `catch()` blocks: catching a class means catching *all of the parent classes* as well.

3.3 Finalize the exception handling

You can notice, that there is a `finally` option as well, that will run in every case. It is very important that codes in the `finally` block can not fail (can not throw any exception)! This section will run in every case, regardless of the existence of an exception, or if it was caught or not. The finally block must be the last section of a `try ... end_try` block.

3.4 Closing the `try` block

You must close the `try` block sequence with an `end_try;` statement. You don't have to use all of the possible blocks in the `try ... end_try` block, however at least one `catch` or `catch_any` or `finally` block must be used. Every executed `try` must have an executed `end_try`! In practice this means that you must NOT `return` from within the `try` block! (Or jump out with `goto`, but who does use it? :-))

3.5 Protecting dynamic memory blocks and objects

Unlike C++ there is no stack unwinding during the exception handling! Consequently you must pay extra attention on memory handling in your routines: make sure that every temporarily allocated memory block is freed in case of an exception rises in the routine or in routines called. The simplest solution is a `try ... finally ... end_try` block. It is relatively not computationally expensive, and is effective. For example the code below will lead to memory leak if there would arise an exception:

```
void my_func( void )
{
    char * mem;

    mem = ooc_malloc( 1000 );

    do_a_risky_call(); /* If this code throws an exception then  */
                       /* the mem will never be freed, causing a */
    ooc_free( mem );   /* memory leak!                           */
}
```

The correct solution is protecting the sensible variables like follows:

```
void my_func( void )
{
    char * volatile mem = NULL;

    try {
        mem = ooc_malloc( 1000 );

        do_a_risky_call();
        }
    finally {
        ooc_free( mem );
        }
    end_try;
}
```

Listen to the followings in the above code:

- Use `volatile` storage class specifier for those variables that change their value in the `try` code section, and you would like to use this new value in any of the `catch` or `finally` blocks, or after the `end_try` statement! This is necessary, because the `try` solution is based on `setjmp/longjmp`, that may change the register values, so we must prevent optimizing compilers using registers for these variables. Forgetting setting `volatile` typically brings you in a situation where the debugged code works properly, but the optimized release fails while handling exceptions.

- Initialize the pointer variable with `NULL`! This is necessary because the local variable is located on the stack, and gets a random starting value. If you forget the initialization, and the memory allocation would fail, then freeing this pointer in the `finally` section would refer to an undefined memory block, and most probably would cause a segmentation fault.

In most cases you want to prevent memory leak only, and do not necessarily need to get the control in the case of an exception. For those situations there is a simpler mechanism described in the next section.

3.6 Managed pointers

In **ooc** you have an other option for preventing memory leaks in case of an exception: the managed pointers. Using managed pointers you will not get the program control in case of an exception, but it is guaranteed, that the memory is freed or the Object is deleted in case of an exception. (You may consider this as analogie for `std::auto_ptr<>` in C++.)
Using managed pointers is faster than using the `try ... finally ... end_try` constructs, so it is more advisable if you do not need the program control in case of an exception.

3.6.1 Managing a pointer

Managing a pointer means that **ooc** will take care of freeing the resouce in case of an exception. You can manage a pointer with the `ooc_manage()` macro. This macro pushes the pointer (and the corresponding destroyer function) to the top of the managed pointers stack.
If there is an exception thrown, **ooc** will continue the program execution at the next `catch` or `finally` statement, and takes care that all memory or Objects that are referenced by the managed pointers pushed onto the stack, are freed or deleted respectively till that point.
If there was no exception thrown, you must remove the pointer from the stack with the `ooc_pass()` macro.
Because the managed pointers' stack is a *stack*, you can remove the most recently pushed item only: you must use `ooc_pass()` always in the reverse order of using `ooc_manage()`!
Use `ooc_manage()` / `ooc_manage_object()` and `ooc_pass()` always as a pair in the same name scope! These macros use local variables, and the variable created by `ooc_manage()` must be accessible by `ooc_pass()`! Never let `ooc_manage()` be executed more than once without executing the corresponding `ooc_pass()` before!
To be ANSI-C compliant, the `ooc_manage()` and `ooc_manage_object()` macros always must preceed any statement in the given namespace! (This is because they define a local

variable.) In practice this means that you must open a new name scope with `{` if you'd like to use the managed pointer in the middle of your code. Close this name scope only after passing the managed pointer. You can nest more namescopes when using multiple managed pointers. See the first example!

3.6.1.1 Manage a pointer: `ooc_manage()`

Pushes a pointer onto the top of the managed pointers' stack.
`ooc_manage()` requires two parameters: the pointer to the resource and the appropriate destroyer function for it (typically `ooc_delete` or `ooc_free`).
`ooc_manage()` does not return anything.

3.6.1.2 Manage an Object: `ooc_manage_object()`

Manages an Object like `ooc_manage()`.
This is shortcut for `ooc_manage( my_object, (ooc_destroyer) ooc_delete );`.

3.6.1.3 Pass the ownership: `ooc_pass()`

Removes the most recently pushed pointer from the managed pointers' stack. Always use in the reverse order of using `ooc_manage()`!
`ooc_pass()` requires one parameter: the pointer to be removed.
Please note that since the most recently pushed pointer is removed, the parameter is used only for verification that the push an pop orders are correct! (In release versions this verification is skipped for gaining some speed.)
`ooc_pass()` returns the pointer itself.
The name is coming from passing the ownership of the pointer to an other object or function.

3.6.2 Examples

3.6.2.1 Protecting temporary memory allocation

In the previous section we used `try ... finally ... end_try` to prevent memory leak for a temporary memory allocation. The same with managed pointer:

```
void my_func( void )
{
    char * mem;

    mem = ooc_malloc( 1000 );
    {
        ooc_manage( mem, ooc_free );

        do_a_risky_call();

        ooc_free( ooc_pass( mem ) );
    }
}
```

Simplier, faster.

3.6.2.2 Taking over the ownership of parameters

```
void foo_add_bar( Foo self, Bar bar )
{
    ooc_manage( bar, (ooc_destroyer) ooc_delete );

    do_a_risky_call();

    /* pass the ownership of bar to self  */
    self->bar = ooc_pass( bar );
}

Foo foo;

foo = foo_new();

foo_add_bar( foo, bar_new() ); /* this code is safe! */
```

If you we're not using managed pointers for taking over the ownership of the parameter then the parameter object would be leaked in case of an exception in the `do_a_risky_call()` method.

4 Using Classes

4.1 Initializing the class

Before you can use your class, you must initialize it! Initializing a class will automatically initialize its superclasses, so if you have inherited your class from occ system classes (like RefCounted), you do not have to initialize them separately. However, because **ooc** uses the `Exception` class internally, you must initialize it before calling any ooc function.

```
int
main( int argc, char * argv[] )
{
    ooc_init_class( Exception );
    ooc_init_class( MyClass );

    do_my_program();

    return 0;
}
```

4.2 Creating an object of a class

Creating an object is easy with the `ooc_new` marco, or with the equivalent `ooc_new_classptr` function.

The `ooc_new( classname, void * param )` macro converts the Class name to the appropriate class description table address, and calls the `ooc_new_classptr` function. Use `ooc_new` when you create your class from a statically known class (you know the class name). The second parameter is passed to the class constructor code without any modification or check. You can parameterize your constructor this way, it is advisable passing a variable or struct pointer here. The variable or the struct must exist until the constructor returns!

The `ooc_new_classptr( Class class_ptr, void * param )` function creates an object of the class pointed by the first parameter. Use this function when you know only the class description table's address! This is very rare situation, and I guess it is mainly useful inside the **ooc** code. The second parameter is passed to the class constructor code without any modification or check.

The above methods return objects of `Object` type.

Although using the `ooc_new` macro for object creation is easy, it is advisable to define a parameterized `..._new()` function for each class, because that way you can control the parameter checking, as well as the automatic conversion of return type (`ooc_new` returns `Object` that you must cast to the desired type).

```
String str_new( char * initial )
{
    return ooc_new( String, initial );
}
```

4.3 Deleting an object

The created objects must be deleted with one of the deletion functions. They *must not* be freed with the standard memory handling functions, like `free` or `ooc_free`!

4.3.1 Deleting an object directly

Deleting an object can be done with the `ooc_delete( Object )` function. It calls the class destructors on the object, and frees the allocated memory. Any pointer to this object will not be usable after deleting the object! Use this way an object destruction when you would like to destroy objects that you did allocate temporarily in your C functions as local variables.

4.3.2 Deleting object via pointer

In many cases it is important to mark that the object has been deleted by nulling the pointer. But this operation rises some issues regarding circular object references, destroying non-complete ojects and multi-threading.

For your convenience there is the object deletion function via its pointer, `ooc_delete_and_null( Object *)`, which does exactly the same in a thread-safe (or at least reentrant) way.

```
void String_destructor( String self, StringVtable vtab )
{
    /* Deletes the member object and nulls the pointer */
    ooc_delete_and_null( & self->other );
}
```

Always use this way a object destruction when you would like to destroy objects that you did allocate globally or as members of other classes! Especially it is important in class destructor codes! (Since ooc 1.0, it is guaranteed that the destructor runs only once. However it is still recommended nulling pointers in your code when deleting, just for clarity.)

4.4 Accessing class members

The class members can be accessed via the object pointer if they are made public, although it is not an advisable method. It is recommended accessing the class members only via the class methods.

4.5 Finalizing a class

A class can be finalized when no longer needed, although it is not necessarily required. If your class has reserved some global resources, then you can release them in the class' finalize code. One may neglect finalizing, if known, that the class did not allocate any global resources. However it is a good practice to finalize the classes that are not needed in the future, especially in dinamically loadable modules.

You can finalize all initialized classes:

```
ooc_finalize_all( );
```

as the last executed line in your code. In case of `ooc_finalize_all()` the class finalization is done in the reverse order of initialization.

However `ooc_finalize_all()` is the preferred way, you can finalize a given class, like:

```
ooc_finalize_class( String );
```

But be very carefull, here! If you finalize a class that would be required for using or finalizing an other class then your code will crash! `ooc_finalize_all()` keeps track of class dependencies, so this problem does not exist that case.

4.6 Dynamic type checking

ooc provides dynamic type safety for your objects, because the object types are known in run time. You can safely cast `Object`s using the `ooc_cast` macro. If the cast fails then `err_bad_cast` exception is thrown. An example of using the dynamic cast in ooc:

```
DeclareClass( String, Base );
DeclareClass( Utf8, String );
DeclareClass( Something, Base );

Something something = something_new();
Utf8   my_utf8_string = utf8_new( "This is an utf8 string." );
String my_string;

my_string = ooc_cast( my_utf8_string, String ); /* Correct */
my_string = ooc_cast( something, String );      /* Can not be cast,
                                                   exception is thrown */
```

You can also retrieve the type of an `Object`. The `ooc_get_type` function returns the type of an object in run time (actually returns a pointer to the class description table). If the parameter is not a valid `Object` then `err_bad_cast` exception is thrown. An example of retriving of the object type in ooc:

```
String my_string = string_new( "Test string." );

printf( "The type of my_string is: %s\n", ooc_get_type( my_string )->name );
```

This example prints `The type of my_string is: String` on the display.

You can use this function for comparing object types, like

```
if( ooc_get_type( my_object ) == & StringClass )
    ... process my_object as a String object here
```

```
else if ( ooc_get_type( my_object ) == & SomeOtherClass )
        ... process my_object as a SomeOther object here
```

Please note the `Class` suffix to the object's typename. This pointer to the static class description table is defined for each class.

But there is a more convenient way for dynamic type checking in ooc, that handles the class inheritance correctly. The `ooc_isInstanceOf` macro returns `TRUE` if the object is an instance of a given class or of any of its superclasses, `FALSE` otherwise. The typical use of this kind type checking is at the very beginning of the class methods, to make sure that the parameter object is of the right type. Because this is a bit computationally "expensive" operation, it is usually placed into an `assert` macro, that lets you some control distinguishing debug and release versions.

```
void
string_example_method( String self )
{
    assert( ooc_isInstanceOf( self, String ) );

    /* Do your method here!
       You can be sure, that self is a valid
       instance of String class.
       (At least in the debug version!)
    */
}
```

5 Implementing Classes

Implementing a new class is relatively simple. You have to create three (or two if you prefer) source files, preferably with the class name. Then you can use the class implementation macros defined in `ooc.h`, and must define the predefined mandatory class methods.

5.1 Naming conventions

Although it is totally up to you, and has no effect on the operation of the **ooc** toolkit, I recommend using the following naming conventions:

- The class name should be relatively short, let's say maximum eight-twelve characters.
- For file names use only lower case letters, like `foo.h`
- Use capital first letter for class names and as the object type, like `Foo`.
- Use a lower case initial for every class member function, making evident that this member function belongs to this class, like `foo_new`.
- You should not apply any initials for the class member names and virtual function names. They will be accessed unambigously.

5.2 Source files

The following files must be created for implementing class Foo that is a subclass (child) of Parent:

- Class user header file. This contains the declarations of class Foo that are used by the users of the class.
- Class Implementation header file. This file contains all declarations that are protected for class Foo.
- Class implementation file. This file contains the declarations private to class foo and the class methods.

5.3 Class user header file

This file should be named as `foo.h`.

In `foo.h` you must decalare the class and its virtual functions, plus the public methods of the class.
You always have to use the virtual function definition block, even you class do not have any virtual function. In this case just leave this block empty.

```
#ifndef FOO_H
#define FOO_H 1

#include "parent.h"

DeclareClass( Foo, Parent );

Virtuals( Foo, Parent )
```

```
EndOfVirtuals;

/* Foo methods */

Foo foo_new( int initial_value );

int foo_get_value( void );

#endif
```

Please note that there is no semicolon after the `Virtuals`.

5.4 Class implementation header file

The class implementation header file contains the definitions for data members of the class `Foo`. It is your choice if you creat a separate class implementation header, or you include this section in the `foo.h` as weel.

Including the implementation related definitions in the class user header file you make all class members public; in other word the user of class `foo` can access all data members simple via pointers.

Including the implementation related definition in a separeted class implementation header (e.g. called `impl_foo.h`) you make all data members protected; in other words the user of the class can not access it, but the subclasses always can.

Making really private members would be a bit complicated, and not supported by the macros. (See "pimpl" or "fast pimpl" idioms for a possible solution!)

The content of `impl_foo.h` should look like:

```
#ifndef IMPLEMENT_FOO_H
#define IMPLEMENT_FOO_H 1

#include "impl_parent.h"
#include "foo.h"

ClassMembers( Foo, Parent )

    int    data;
    void * data_ptr;

EndOfClassMembers;

#endif
```

5.5 Class implementation file

In the class implementation file you must allocate the class description table and the virtual table of the class. Then you must implement the mandatory class member functions as below. After this mandatory section you can implement your class methods.

The class implementation file may be called e.g. `foo.c`, but it can consist multiple files if necessary, of course.

5.5.1 Class allocation

```
#include "impl_foo.h"

AllocateClass( Foo, Parent );
```

5.5.2 Class initialization

The most of the class properties are initialized in compilation time. However the vtable can not be initialized perfectly, so initializing a class means building up the class's virtual table.

You must initialize the virtual table only if your class defines *new* virtual functions; or you would like to override any virtual function of the parent class! If you don't have to do anything in the class initialization, just leave its body empty!

The mandatory function name for the class initialization function is the class name `+` the suffix of `"_initialize"`.

This function has got a pointer to the class description table as parameter. You can access the class's virtual table via this pointer. The virtual table address is stored in the `vtable` field of the class description table, and the type of the virtual table is the class name concatenated with `Vtable`.

Example: overriding the parent's `print` virtual function:

```
static
void
Foo_initialize( Class this )
{
    FooVtable virtuals = (FooVtable) this->vtable;

    virtuals->Parent.print = virtual_foo_print;
}
```

Example: aquiring some global resources in the class initialization code:

```
static List foo_list = NULL;

static
void
Foo_initialize( Class this )
{
    ooc_init_class( List ); /* make sure, that List has been initialized */

    foo_list = list_new( ooc_delete );
}
```

You can call `ooc_init_class( ClassName )` as many times, you need, the `ClassName_initialize( Class )` function will be called only once. (Until `ooc_finalize_class( ClassName )` is not called.)

You can throw exception in `ClassName_initialize( Class )` function.

5.5.3 Class finalization

If you have aquired some global resources during class initialization, you may want to release them before exiting your program. The class finalization method is there for this purpose. The class finalization must not throw an exception!

```
static
void
Foo_finalize( Class this )
{
    ooc_delete_and_null( & foo_list );
}
```

It is guaranteed, that `ClassName_finalize( Class )` is called only once for each `ClassName_initialize( Class )`. In most cases the class finalization is just a simple empty function, doing nothing.

5.5.4 Constructor definition

The constructor is responsible for building up an object of the class. The constructor has a fix name: the class name concatenated with `_constructor`.

In the constructor you can be sure, that all data members are set to 0 (or NULL in case of a pointer) prior calling the constructor.

If your class has a parent class (other than `Base`) then the *first thing in a constructor is calling the parent class's constructor* using the `chain_constructor()` macro! It is advisable putting the `chain_constructor()` macro always at the begining of your con-

structor, because this practice makes the task of changing the inheritance more easy. The `chain_constructor()` macro has three parameters:

- Name of your *actual* class,
- The actual object pointer,
- Parameters for the parent constructor.

The class constructor has two parameters: the address of the object itself as an `Foo` object, and a pointer to the parameters. This parameter pointer was the second parameter of the `ooc_new()` function, or was assigned by the subclass constructor by the `chain_constructor()` macro.

```
static
void
Foo_constructor( Foo self, const void * params )
{
    assert( ooc_isInitialized( Foo ) );   /* It is advisable to check if the class has
                                             been initialized before the first use */
    chain_constructor( Foo, self, NULL ); /* Call parent's constructor first! */

    self->data = * ( (int*) params );
}
```

If you encounter any problem in the construction code, you can throw an exception here.

It is advisable defining a convenient wrapper around the `ooc_new()` call to make the parameter type checking perfect and being able to aggregate multiple parameters into a single parameter struct, that can be forwarded to the `ooc_new()` as the second parameter, and not less importantly converting the returned `Object` type automatically to your specific object type.

```
Foo
Foo_new( int initial_value )
{
    return (Foo) ooc_new( Foo, & initial_value );
}
```

5.5.5 Copy constructor definition

The copy constructor creates a second object of your class. The `ooc_duplicate` uses this constructor when creating a duplicate of the class.

The copy constructor has a fix name: the class name concatenated with `_copy`.
The copy constructor has two parameters: a pointer to the new object, and a pointer to the object that is copied.
The copy constructor must return:

- `OOC_COPY_DONE`, if you have copied the object succesfully,
- `OOC_COPY_DEFAULT`, if you have not copied anything, and the default copy must be applied,
- `OOC_NO_COPY`, if this object can not be copied.

When entering into the copy constructor you can be sure that all the parent class' members are already copied succesfully, and all class members are set to 0 or NULL.

If you encounter any problem in the construction code, you can throw an exception here.

5.5.5.1 Using the default copy constructor

If your class do not require any special action when it is copied (the bit-by-bit copy is OK) then you can leave all the task to the class manager, by simply returning `OOC_COPY_DEFAULT`:

```
static
int
Foo_copy( Foo self, const Foo from )
{
    /* makes the default object copying (bit-by-bit) */
    return OOC_COPY_DEFAULT;
}
```

But be careful with the default copying! Copying pointers may lead unexpected double frees of memory block and may crash! Make your own copy, if you have pointers, reference counted pointers, etc.!

An other aspect is the performance. Because the default copy uses the `memcpy()` for completing the copy of an object, it is a bit "expensive", it has too much overhead. If your program is using `ooc_duplicate()` extensively, it is recommended creating your own copy constructor for smaller objects.

5.5.5.2 Creating your own copy constructor

Creating your own copy constructor is simply, and mostly self-explanatory.

```
static
int
Foo_copy( Foo self, const Foo from )
{
    self->data  =  from->data;

    return OOC_COPY_DONE;
}
```

Do not forget to *return OOC_COPY_DONE*, otherwise the default copy will run and will overwrite everything that you made!

5.5.5.3 Disabling the copy constructor

Unfortunately it is not possible disabling the copy constructor in compilation time, like in C++. (In C++ this is the technique making the mandatory copy constructor `private: Foo::Foo( Foo& )`, so noone will be able to access it.)
However you can prevent copying the object in runtime, simply returning `OOC_NO_COPY`, that forces throwing an `Exception` with the `err_can_not_be_duplicated` error code.

```
static
int
Foo_copy( Foo self, const Foo from )
{
    return OOC_NO_COPY;
}
```

5.5.6 Destructor definition

The destructor destroys the object of your class before releasing the allocated memory. The `ooc_delete` uses this destructor when deleting an object.

The destructor has a fix name: the class name concatenated with `_destructor`.
The destructor has two parameters: a pointer to the object to be detroyed, and a pointer to its virtual table.

Within the destructor you can *not* throw any exception!
In the destructor you must consider, that your object is not valid: the virtual table pointer was nulled before entering in to the destructor. This is for marking the object that deletion is pending, and preventing multiple entry into the desctructor. (This way we could save some bytes in each objects.) This means that you can not use `Virtual` macro, or other macros that use the virtual table, e.g. `ooc_isInstanceOf()`.
However you can still access the virtual functions via the `vtab` parameter, so you can use them if you need. Since ooc 1.0 it is guaranteed that the destructor runs only once. However you should use only `ooc_delete_and_null()` and `ooc_free_and_null()` in destructors! This prevents crashes because of double freeing or deleting in case of circular references.

```
static
void
Foo_destructor( Foo self, FooVtable vtab )
{
    ooc_free_and_null( & self->data_ptr );
}
```

5.5.7 Implementing class methods

The class methods are normal C functions with the first parameter as a pointer to the object.

Because there is no real parameter type checking in C when calling this class method, it is possible to pass *anything* to the class method as its first parameter! This is error prone, so it is a good practice to always check the first parameter within the class method!

5.5.7.1 Non-virtual methods

Non-virtual methods are global C functions.

```
void
foo_add_data( Foo self, int size )
{
    assert( ooc_isInstanceOf( self, Foo ) );

    self->data_ptr = ooc_malloc( size );
}
```

5.5.7.2 Virtual methods

Virtual methods are static C functions, that are accessed via pointers in the virtual table.

See section "Virtual Functions" for more information!

5.6 Classes that have other classes

You can have classes that embody other classes. You may implement them as normal objects, and use `ooc_new()` in the outer objects constructor, to allocate and build the related object, like:

```
ClassMembers( Foo, Base )
    Bar    bar;
EndOfClassMembers;

    ....

static
void
Foo_constructor( Foo self, const void * params )
{
    chain_constructor( Foo, self, NULL );

    bar = ooc_new( Bar, params );
}

static
void
Foo_destructor( Foo self, FooVtable vtab )
{
    ooc_delete_and_null( (Object*) & self->bar );
```

```
}
```

In this example Foo object can be considered, that it *has* a Bar object as a member. But this way of constructing the Foo object is not effective, because there are two memory allocations: one for Foo and the other for Bar in Foo's constructor. This requires more time, and leads to more fragmented memory. It would be a better idea to include the body of the Bar object completly into the Foo object. You can do it, but must take care, that you must use `ooc_use` and `ooc_release` instead of `ooc_new` and `ooc_delete` respectively, because there is no need for additional memory allocation and deallocation for the Bar object!
The above example rewritten:

```
ClassMembers( Foo, Base )
    struct BarObject    bar;
EndOfClassMembers;

    ....

static
void
Foo_constructor( Foo self, const void * params )
{
    chain_constructor( Foo, self, NULL );

    ooc_use( & self->bar, Bar, params );
}

static
void
Foo_destructor( Foo self, FooVtable vtab )
{
    ooc_release( (Object) & self->bar );
}
```

Less malloc(), better performance!
Of course, in this case you can access the member of the included Bar objects a bit different: instead of `self->bar->data` you must reference as `self->bar.data`.

Never use the object inclusion for reference counted objects! The reference counting will not work for included objects!

6 Interfaces and multiple inheritance

Since version 1.3 **ooc** introduces a kind of multiple inheritance with the help of interfaces. The idea behind is a bit a mix of the Java and C++ interfaces, but differs in the way of inheritance and the use.

6.1 What is an interface?

An interface is simply a collection of functions that describe the behavior of the Object in some aspect. The interface itself does not implement any functionality, it just defines what methods the Object must have, and behave according to it. In some design methods this is called a contract for the Object. The Object should implement its own implementation of the contract: this is called impementing the interface. In Java the interface is a pure abstract class without a data member, in C++ it is called a pure virtual class without constructor, destructor and data members.
The easiest way to define a group of functions is to collect some function pointers in a C struct. For example in pure C we would write:

```
struct DrawableInterafce
{
    void     (* draw)( Object );
    void     (* rotate)( Object, double );
    int      (* get_heigth)( Object );
    int      (* get_width)( Object );
};
```

This describes the behavior of a drawable `Object`. Any Object type that implements a `DrawableInterafce` can be asked for its dimensions and can be drawn and rotated.

6.1.1 Interfaces and inheritance

The use of interfaces provides a kind of multiple inheritance for **ooc**. While the classes can be inherited in a single inheritance chain (each class can have only one superclass), every class can implement as many interfaces as nedded. Since interfaces can be implemented by unrelated classes, it is a kind of multiple inheritance, like in Java.
In **ooc** interfaces are simply added to the virtual table, so you can reference and use them as any other virtual function of the class! This is very similar to the C++ implementation of an interface.
As a consequence, in **ooc** the interface implementation is inherited by the subclasses and can be overriden (like in C++, and unlike Java).
The good news is that by the help of interfaces **ooc** introduces multiple inheritance. The bad news is, that multiple inheritance calls for the dread problem of diamond inheritance (http://en.wikipedia.org/wiki/Diamond_problem).
Since an ANSI-C compiler has nothing to handle such a situation, we must avoid any possibility of a diamond inheritance, thus the interfaces themselves can not be inherited (other words: can not be extended) in **ooc**.
The following table summarizes the behaviour of interfaces in different languages:

	ooc	C++	Java
The implementation of the interface is inherited by subclassing a class	yes	yes	no
Interface methods can be overriden in subclasses	yes	yes	must be
The interface itself can be inherited (a.k.a. extended in Java)	no	yes	yes

6.1.2 Creating an interface

Creating an interface is very easy in **ooc**: you just simply deaclare its members in a publicly available header file. For example we create an interface for a drawable Object:
In `drawable.h`:

```
DeclareInterface( Drawable )

    void     (* draw        )( Object );
    void     (* rotate      )( Object, double );
    int      (* get_heigth  )( Object );
    int      (* get_width   )( Object );

EndOfInterface;
```

Please note, that the first parameter for each method in the interface is `Object`! This is, because any kind of a `Class` can implement the interface, so we can not limit the type of the `Object` for just a given class.
Each interafce has a unique identifier, an interface descriptor that must be statically allocated (in ROM on microcomputers). This can be done in `drawable.c`, or if we have many interfaces, collected them together in e.g. `interfaces.c`:

```
#include "drawable.h"

AllocateInterface( Drawable );
```

That's it! We have done: the interface is declared and has a unique id: `Drawable`.
To minimize your work you can use the ooc tool to create the skeleton:

```
~$ ooc --new Drawable --template interface
```

6.1.3 Implementing an interface

To implement an interface for a class, you must do the followings:

1. Add the interface to the class's virtual table.
2. Implement the interface methods for the class
3. Initialize the virtual table with the interface method implementations
4. Register the implemented interface(s) for the class
5. Allocate the ClassTable using the interface register

So let's say we implement the Drawable interface for the `Cat` class! In the followings we show only the steps necessary to implement the interface, other issues covered earlier are

not repeated here!
Implelementing multiple interfaces for the class is the same, just repeat the necessary macros and lines! As an example, along with the `Drawable` interface we add the `Pet` interafce as well.

6.1.3.1 Adding the interface to the virtual table

In the public header (*cat.h*) locate the `Virtuals`, and add the interface with the `Interface` macro:

```
Virtuals( Cat, Animal )

    void     ( * miau )( Cat );

    Interface( Drawable );
    Interface( Pet );

EndOfVirtuals;
```

6.1.3.2 Implementing the interafce methods

The interface method impementations are `static` functions in the class implementation file. This is exactly the same aproach to the implementation of any virtual method. (Do not forget: an interface is just a collection of virtual functions in ooc.)
So in *cat.c* write:

```
static
void
cat_rotate( Cat self, double arcus )
{
    assert( ooc_isInstanceOf( self, Cat ) );

    // Rotate your cat here :-)
}
```

6.1.3.3 Initializing the virtual table

In the class initialization code you must assign the the implementation methods with the appropriate function pointers in the virtual table. Again, see the section about virtual functions.

```
static
void
Cat_initialize( Class this )
{
    CatVtable virtuals = (CatVtable) this->vtable;

    virtuals->Drawable.rotate = (void (*)(Object, double)) cat_rotate;
    // add the other methods as well ...
}
```

6.1.3.4 Registering the implemented interafces

There is a way to retrieve the implemented interfaces for any class, that is the `ooc_get_interface()` function. To let it work, we must register all implemented interfaces for the class. This is done in the class implementation file, preferably just right before the class allocation. So, in `cat.c` we register the Drawable and Pet interfaces for the Cat class:

```
InterfaceRegister( Cat )
{
    AddInterface( Cat, Drawable ),
    AddInterface( Cat, Pet )
};
```

Listen to the different syntax! Internally this is a table of structs, so you must end it with a semicolon, and use comma as the internal list separator. You must not put a comma after the last item in the list!
`ooc_get_interface()` scans this table, so you can get some increase in speed of average execution, if you place your most frequently used interfaces in the first positions.

6.1.3.5 Allocating the class with interfaces

The only thing remained is to let the ClassTable know about the interface register. So instead of `AllocateClass` use `AllocateClassWithInterface`:

```
AllocateClassWithInterface( Cat, Animal );
```

If you forget this step, `ooc_get_interface()` will always return `NULL`, since it will not know about the registered interfaces for the class.

With this step we have finished the implementation of an interface for a class.

6.1.4 Using an interface

When we need to invoke an interface method, simply call the virtual function via the function pointer in the virtual table. There can be two situation in practice: we may know the type of the object we have, or may not.

6.1.4.1 If the type of the Object is known

If we know the type of the object we have, and this type has implemented the interface that we need, then we can simply call the virtual function of the interface.

```
Cat mycat = cat_new();

// rotating mycat:
CatVirtual(mycat)->Drawable.rotate( (Object) mycat );
```

This is the fastest way to invoke an interface method, since this is a dereferencing of a function pointer only.

6.1.4.2 If the type of the Object is unknown

The main use of an interface is when we do not know the exact type of the object we have, but we want it to behave according its interface method. The `ooc_get_interface()` method can retrieve a given interface from any kind of `Object`. The `ooc_get_interface()`

function returns a pointer to the interface methods of the class, and the desired method can be called via this function pointer.

```
void
rotate_an_animal( Animal self )
{
    Drawable drawable = ooc_get_interface( (Object) self );

    if( drawable )
        drawable->rotate( (Object) self );
}
```

`ooc_get_interface()` returns `NULL` if the desired interface is not implemented for the object. If you use this function, always check for a `NULL` pointer, otherwise you will generate a segmentation fault, if the interface is not implemented.
In case you know that your object has implemented the desired interface, but you do not know the exact type of your object, you can use the `ooc_get_interface_must_have()` function instead. This will never return a `NULL` pointer, but throws an `Exception` if the object does not have the desired interface.

```
ooc_get_interface_must_have( (Object) self )->rotate( (Object) self );
```

Note: the `ooc_get_interface()` function traveses the InterfaceRegister for the class and its superclasses, so it may be relative slow. If you use the interface for an object multiple times within a context, it is worth to retrieve the interface pointer only ones for the object, and keep it around until needed.

6.2 Mixins

The mixins are interfaces that have default implementations and have their own data (http://en.wikipedia.org/wiki/Mixin).

A mixin does not know anything about the enclosing Object / Class!

From the user point of view a mixin looks identical to an interface! See [Using an interface], page 29.

A mixin has its own data, but it can not be instantiated directly. It is always held by an Object (a carrier object).

A mixin has the following features:

- May have default implementation of the interface methods (usually has, but this is not mandatory)
- Has its own data (that is accessible by the enclosing class as well).
- The default implementation of the interface methods can be overriden by the enclosing class.
- May have initializing and finalizing, thus can use global resources.

6.2.1 Creating a mixin

The mixin is nothing else than an interface impementation plus a data structure. But this data structure is handled like a class, with very similar methods: there are initialization,

finalization, construction, destruction and copy methods like for a normal Class, and an additional one: the ..._populate() method.
In this section we go through the steps creating a mixin.
To create the mixin skeleton for `Flavour`, just type:

```
~$ ooc --new Flavour --template mixin
```

Three files has been created:
The *`flavour.h`* is familiar, it looks like a normal interface declaration except it is declared with `DeclareMixinInterface()`.
Put your interafce methods (function pointers, of course) into the declaration as needed.

There is a *`implement/flavour.h`* file, that contains the data member declaration for the mixin. Add your data members as needed.
These data fields will be mixed into the enclosing class's object structure, and can be referenced via a `FlavourData` type pointer (interface name `+ Data` suffix).

In the *`flavour.c`* file you will find the necessary `AllocateMixin` macro, and you must provide the method implementations for the mixin. The following methods must be implemented:

`Flavour_initialize()`
: This is called only once, and is called automatically before the initialization of the first enclosing class.
Shared or global resources used by the mixin may be initialized here.

`Flavour_finalize()`
: Called only once when `ooc_finalize_all()` is called.
Release the shared or global resources that were initialized by the mixin.

`Flavour_constructor( Flavour flavour, FlavourData self )`
: Initializes the data fields of the mixin. All data fields are garanteed to be zero at entrance.
This method is called automatically before the enclosing class's constructor is called.
If your data fields do not require any construction, you can leave this method empty.
The interface methods are set (and are overriden by the enclosing class), so they can be used here. (The first parameter.)
Note, that there are no constructor parameters! There is only "default constructor" for mixins. If you need some values or construction depending on the enclosing object then implement a setter or constructor method in the interface itself and call it from the enclosing class's constructor!

`Flavour_destructor( Flavour flavour, FlavourData self )`
: Destroys the data fields of the mixin.
This method is called automatically after the enclosing class's destructor, and the normal rules within a desctructor must be applied here as well!
If your data fields do not require any destruction, you can leave this method empty. The interface methods are set (and are overriden by the enclosing class),

so they can be used here. (The first parameter.)

`Flavour_copy( Flavour flavour, FlavourData self, const FlavourData from )`
: Copy constructor. This is very similar to the copy constructor used for `Objects`!
Must return: `OOC_COPY_DEFAULT`, `OOC_COPY_DONE` or `OOC_NO_COPY`, and the same rules apply!
This method is called automatically before calling the enclosing class's copy constructor.
The interface methods are set (and are overriden by the enclosing class), so they can be used here. (The first parameter.)

`Flavour_populate( Flavour flavour )`
: Populates the interface method table with function pointers to the default mixin method implementations.
This method is called automatically before calling the enclosing class's initialization method.

The mixin interface methods' implementation must follow an important rule! Since the mixin does not know where its data is located, the mixin's data must be retrieved first! The only thing that a mixin method knows is the carrier `Object` (always the first parameter for an interface method!). Retrieving the mixin's data fields can be done with the `ooc_get_mixin_data()` function.
Therefore a typical mixin interface method starts with the followings:

```
static
void
flavour_set( Object carrier, const char * flavour_name )
{
    FlavourData self = ooc_get_mixin_data( carrier, Flavour );

    self->name = flavour_name;
}
```

6.2.2 Implementing a mixin by a carrier class

The implementation of a mixin for a class is almost the same as the implementation of an interface, with few additional tasks.
The steps to implement some `Flavour` for the `IceCream` class are:

- Create the `IceCream` class with *`~$ ooc --new IceCream`*.
- Add the `Interface( Flavour );` macro to the `IceCream` virtuals in *`icecream.h`*.
- Add the `MixinData( Flavour );` data member to the end of the `ClassMembers( IceCream, Base )` structure in *`implement/icecream.h`*.
  It is very important that no data members can follow the `MixinData()` macros in the `ClassMembers()` list, otherwise the copy constructor will not work correctly!
- Create the Interface register for `IceCream` in *`icecream.c`*.

```
InterfaceRegister( IceCream )
{
    AddMixin( IceCream, Flavour )
};
```

- Replace the `AllocateClass( IceCream, Base );` macro with `AllocateClassWithInterface( IceCream, Base );` in *icecream.c*.
- Override the necessary `Flavour` interface methods as needed in `IceCream_initialize()`.
- Set some initial vale for the `Flavour` mixin while constructig an `IceCream` object, if the default constructor is not sufficient for you.

```
static void IceCream_constructor( IceCream self, const void * params )
{
    assert( ooc_isInitialized( IceCream ) );
    chain_constructor( IceCream, self, NULL );

    IceCreamVirtuals( self )->Flavour.set( (Object) self, "vanilla" );
    ...
}
```

6.2.3 How mixins work?

Mixins work like a superclass for the enclosing class: you do not have to care with it, **ooc** does everything automatically for you:

- The mixins are automatically initialized before initializing the enclosing class, just like a superclass.
- The mixin's interface methods in the enclosing class's virtual table are automatically populated with the default implementations before the initialization method of the enclosing class is called.
  (The mixin's `..._populate()` method is called.)
- The mixin's data members are automatically constructed before the constructor of the carrier object is called.
  Using the mixin in the carrier object's constructor is legal.
- The mixin's data members are automatically copied before the copy constructor of the carrier object is called.
  Using the mixin in the carrier object's copy constructor is legal.
- The mixin's data members are automatically destructed after the destructor of the carrier object is called.
  Using the mixin in the carrier object's destructor is legal.
- The mixins are automatically finalized when `ooc_finalize_all()` is called.

This is the same behavior of a superclass, you can treat the implemented mixin as a superclass of the enclosing class as well, except you can not cast the types (Mixins do not have type information at all.), and you must access it as an interface via its interface methods.

However, mixins have some drawbacks!
First: it exposes all mixin data members to the enclosing (carrier) class: the enclosing class

can acces the mixin's data directly, and it is up to the programmer's intelligence not to do bad thigs. :-(
Second: using interafces and mixins frequently is a very elegant design, but be aware that in **ooc** it is a bit expensive! Since a C compiler gives no support for such implementations, every lookup, check and conversion is done in run-time, thus much slower than in C++ for example.

7 Memory handling

There are basic memory handling wrappers around the standard C library memory handling routines. The key difference is, that the **ooc** memory handling throws exceptions in case of a system failure, and performs some tasks that are necessary for multi-threaded or reentrant programming.

7.1 Memory allocation

You can allocate memory with the following routines:

```
void * ooc_malloc( size_t size );
```

The same to the standard `malloc`, except that throws an exception on failure, thus never returns `NULL`.

```
void * ooc_calloc( size_t num, size_t size );
```

The same to the standard `calloc`, except that throws an exception on failure, thus never returns `NULL`.

```
void * ooc_realloc( void *ptr, size_t size );
```

The same to the standard `realloc`, except that throws an exception on failure, thus never returns `NULL`.

```
void * ooc_memdup( const void * ptr, size_t size );
```

Duplicates a memory block using the standard `memcpy`, and throws an exception on failure, thus never returns `NULL`.

7.2 Freeing the allocated memory

Deallocate the memory with one of the following methods:

```
void ooc_free( void * mem );
```

Frees a memory block allocated with one of the above allocation codes. Never failes, `mem` can be `NULL`;

```
void ooc_free_and_null( void ** mem_ptr );
```

Frees a memory block via a memory pointer and nulls the pointer simultaneously. It is very important for thread-safe or reentrant codes. It is also very important for freeing memory blocks with circular references. `mem_ptr` can point to a NULL pointer.

Always use this, if your class has a memory pointer as a class member, for example in a class destructor!

7.3 Thread safety

Important! **ooc** is thread safe *only* if the underlying standard C library (malloc, calloc, realloc, setjmp/longjmp) is thread-safe too!

The thread safety of **ooc** does not mean that your code will be thread safe automatically! You must take care about the proper thread-safe implementation of your classes!

The `ooc_init_class()`, `ooc_finalize_class()` and `ooc_finalize_all()` functions are *not* thread safe! It is advisable calling `ooc_init_class()` from the main thread before the fork and `ooc_finalize_all()` after the join.

8 Unit testing support

It is very important testing your classes before you integrate them into a larger part of code. The Unit testing support in **ooc** is inspired by the Unit 3 testing framework used mainly in Java. If you are not familiar with the topic I recommend some reading about the "test-driven development" approach to catch the basis and importance of unit testing.

8.1 How to create a unit test?

Unit test classes are subclasses of `TestCase` class, these are called test cases. A test case in **ooc** is represented by an executable that can be invoked from the operating system or can be run in a simulator in case of microcontrollers.
The test cases have several test methods, that do different tests. These test methods are executed in the order as they are defined in the test method order table.
The test cases could be organized into test suites. The latter means a batch file in **ooc** that executes the test cases in a sequence.
Its recommended collecting your testable classes in a static directory (*.o in Linux, *.lib in Windows, etc.) to make things simplier. Although this is not necessary, this makes the linking of your testcase more simply, and enables you creating fake classes as needed.

Creating a unit test is easy!

1. Make a directory in wich you would like to collect your test cases. This directory will be your test suite directory, since it will hold your test case, your makefiles and test siute batch files. Select this directory as your working folder. (For very small projects this practice may not be necessary, you may use the source directory too for your tests.)
2. Create your test environment with the following command:

   ```
   ~$ ooc --testsuite
   ```

   This creates the necessary makefiles and test suite batch files into this directory. Edit those files according to the comments in them as necessary. If you are working with an IDE, you can configure it to use these makefiles for convenience. It can be used with the automake tools as well under Linux.
3. Create your test case with the following command:

   ```
   ~$ ooc --new MyClassTest --template test
   ```

   As a naming convention I recommend using "Test" as a tail for your test classes. This helps identify them, but also lets you use the automatic makefile rules in Linux! In Windows you must add this file to the Makefile: edit it according to the comments in it.
4. Implement the `before_class()`, `before()`, `after()`, `after_class()` methods in your test file if they are necessary for your test methods. The skeletons can be found in the created file already.
5. Implement your test methods. Add them to the test method order table. For details see the next sections.
6. Build your test case with the supplied Makefile.

7. Run your test case as an executable.
8. Run your testsuite. Edit your test suite file (suite.sh or suite.bat) as required by the help of the comments in it, and start that script/batch file.

8.2 Writing a unit test

8.2.1 Writing test methods

Implement your test method as a function of the following prototype:

```
static void myclasstest_method1( MyTest self );
```

You can expect that the `before()` method has been completed succefully before invoking your test method, and the `after()` method will be invoked after completing your test method.
The test method can throw an `Exception` or a subclass of it. If the exception is uncaught within the method, the test case repots the test as failed. If the `Exception` is not caught within your test method, the `after()` method will not be invoked!

You must put your test method into the test method order table, to let your test case know what methods to invoke! You can do this using the `TEST` macro:

```
static ROM_ALLOC struct TestCaseMethod methods[] =
{
    TEST(myclasstest_method1),

    {NULL, NULL} /* Do NOT delete this line! */
};
```

Important! In ooc your test methods are not independent! This means that if one of your test methods makes changes into any global or test case member variables, or into the environment, the succeeding test methods will inherit those changes. This is a very important difference from Unit 3, where it is garanteed that all test methods starts in a fresh environment.

8.2.2 Assertions

This is why we do the whole staff! :-) Within our test methods we would like to make assertions on the status of our testable classes. For this purpose we can use a group of `assertCondition` macros. (Don't be confused: these macros do nothing with those ones in the <assert.h> headers!) The `assertCondition` macros check the given condition, and if the assertion is not met, print an error message identifying the line of code. The execution of code continues in both cases.
A group of `assertCondition` macros can print messages defined by the user as well. These macros look like `assertConditionMsg`.
An example for using the `assertCondition` macros from the `ListTest` unit test case, testing the list iterator:

```
static
void
iterator( ListTest self )
{
    list_append( self->foolist, foo_new() );

    assertTrue( self->foolist->first == list_first( self->foolist ) );
    assertTrue( self->foolist->last == list_last( self->foolist ) );
    assertTrue( list_first( self->foolist ) == list_last( self->foolist ) );

    list_append( self->foolist, foo_new() );

    assertTrue( self->foolist->first == list_first( self->foolist ) );
    assertTrue( self->foolist->last == list_last( self->foolist ) );
    assertFalse( list_first( self->foolist ) == list_last( self->foolist ) );
    assertTrue( list_last( self->foolist ) == list_next( self->foolist, list_first( se
    assertTrue( list_first( self->foolist ) == list_previous( self->foolist, list_last
}
```

For the complete list of `assertCondition` macros see the **ooc** API documentation, or the `testcase.h` header file.

There is a macro that reports a failure unconditionally, this is the `fail` (or the `failMsg`) macro. It prints the error message unconditionally, and continues. It can be used if the condition to be tested is not a simple assertion, and the code runs on a bad path.

8.2.3 Messages

Normally the test cases do not print any message on the screen: they inform you about the actual method by printing its name, but this info will be deleted if the method was executed succesfully. On a Linux or Windows system there won't be any messages left on the display in case of expected results!

```
~$ ./listtest
~$
```

If some assertions fails, you will see a message similar to this one:

```
~$ ./listtest
[4] ListTest::iterator()
        Failed:  [listtest.c : 325]
Test case ListTest failed: 30/1 (methods run/failed)
~$
```

This could be an example, if our `ListTest::iterator()` method detected an error in the list iterator implementation. Let's see the meaning of the message!

1. The number in angle brackets shows the sequence number of the method: the failed method was the fourth executed method in the test case.

2. `ListTest::iterator()` informs about the name of the method. The part before the period shows the name of the class, the part after the period identifies the name of the failed test method.
3. The next line says that an assertion in this method failed. If we used an `assertConditionMsg` assertion, the message would be displayed here.
4. In the angle brackets the exact location of the failed assertion is displayed: the name of the source code and the line number in the source code. This information helps you find the failed one fast and easily.
5. The last line is a summary for the test case. It shows how many test methods was run in the test case, and how many of them was failed.

In some cases your test method may throw an exception that is not caught by your method. These exceptions are caught by the TestCase class, displayed as follows, and the code execution continues with the next test method.

```
~$ ./listtest
[4] ListTest::iterator()
        Unexpected exception: Exception, code: 1, user code: 0
~$
```

The caught exception was an `Exception` class with code 1. Since this is a core ooc exception, evaluating the `"exception.h"` header it can be seen, this is `err_out_of_memory`, probably our test method run out of memory.

8.2.4 Overriding virtuals

Use the `before_class(), before(), after(), after_class()` methods to prepare your test case for test methods. Similar to Unit 3: `before_class()` and `after_class()` will run only once: when instantiating your test case class, that is a subclass of `TestCase`.
The `before()` and `after()` methods will run for each test method execution. Generally in `before()` you can prepare the necessary classes for your test methods, while in `after()` you should destroy them.
You can use `assertConditionMsg` assertions freely in these virtuals. It might be usefull especially in the `after()` method to check that the test method did not corrupt the integrity of your test class.
You can throw ecxeptions in `before()` and `before_class()` methods, but never throw in `after()`!
For complicated test cases you can override these methods, since those are virtual methods (but do not do this, if you do not understand exactly how and when they are called. Check the source code if you have any doubt!)
The exexcution order of the virtual methods is:

```
For each instantiation of your test case:
{
        Your testcase's parents' constructors
        Your testcase's constructor
        For each testcase_run( TestCase ):
        {
                Your test's parents' before_class() methods
                Your test's before_class() method
```

```
                For each test method in the test method table:
                {
                        Your test's parents' before() methods
                        Your test's  before() method
                        Yor test method
                        Your test's  after() method
                        Your test's parents' after() methods
                }
                Your test's after_class() method
                Your test's parents' after_class() methods
        }
        Your testcase's destructor
        Your testcase's parents' destructors
}
```

8.2.5 Testing exceptions

Test the expected exceptions as follows! Let's check the index overrun behavior of the Vector class:

```
try {
    vector_get_item( self->vector, -1 );   /* Should throw an Exception! */
    fail();                                /* Should never be executed! */
}
catch_any
    assertTrue( exception_get_error_code( exception ) == err_wrong_position );
end_try;
```

8.2.6 Memory leak test

Tests for possible memory leaks is an essential issue in C! Always run your test cases to discover memory leaks, because this helps you make your code robust and reliable!
In **ooc** the recommended way is using **Valgrind** (http://valgrind.org), a memory checker (and much more!) tool available under Linux. (For Windows I do not know such a tool, please infrom me if there is a similar possibility.)
To perform a memory check, run your testcase in **Valgrind**:

```
~$ valgrind --leak-check=yes --quiet ./myclasstest
```

To minimize your headache, I recommend running only error free test cases under Valgrind! :-) Hunting for assertion failures together with memory problems could be a nightmare!

Note I.: In the Linux standard C library there may be some optimizations that are reported by **Valgrind** as a possible memory leak, but in fact they are not an error, especially not yours. If you face with this problem, you can suppress them. To supress them, create an empty test case, run it in **Valgrind** and create a suppress file. For details see the **Valgrind** documentation: "2.5. Suppressing errors" (http://valgrind.org/docs/manual/manual-core.html#manual-core.suppress).

Note II.: ooc converts the Unix signals (SIGSEGV, SIGFPE) into ooc exceptions. This let us handle and display those problems the same way as other exceptions in ooc. Unfortunately **Valgrind** overrides these signal handlers, and as a consequence, your test case can not be run under **Valgrind** if you force emmitting these Unix signals (e.g. intentionally dereferencing a NULL pointer and checking for `SegmentationFault` exception.)

8.2.7 Unit testing techniques:

There are several best practices in Unit testing. Some of them could be used in **ooc** as well.

8.2.7.1 Inherited test cases

If you must write many test cases that requires identical preparation or a special environment, then you may consider creating a test case to make this preparation and which could be the parent for your test cases.
ooc `TestCase` can be inherited!

8.2.7.2 Using fake objects

In this technique you can mimic the behaviour of an object that is not available during the execution of your test case: you simply replace a class with a fake class that behaves identical (or at least as expected), but does not do its job.
Write your fake class in your test suite directory: all the method names must be identical to the original ones! Link your test case as usual, except that you define your fake object file earlier in the parameter line than the library containing your testable classes. The linker will use the symbols in the order they appear in the parameter list: replacing you class with the fake.

In Linux and Windows this is very simple! Just create your fake class with the *`ooc`* tool in the suite directory (you may copy it from your source directory). Do not copy the header file, since it should be the original one! As a naming convention the source file name must NOT end with `Test`, tail it with `Fake` instead.
Modifiy your code in the fake class. The rest is done by the supplied Makefiles! (In Windows, you must add your file to the Makefile first!)

8.2.7.3 Using mock objects

I have no clue in this moment, how to elaborate this technique in C. :-(

8.2.8 Dependency lookup

For effective unit testing you need a design in wich your classes do not rely on their dependencies. Establishing your classes uncoupled helps you reuse them much easier and lets you implement unit testing. (For more infromation on this topic I recommend some googling on dependency injection, dependency lookup and such topics in Java.) Theoretically it is possible writing a cotainer class in **ooc** that could handle the object/class dependencies based on an external descriptor file (e.g. XML), like in some Java implementations. Although it would be possible, it will not be feasible: small microcontollers do not have the power to execute effectively. So I recommend using the "dependency lookup" idiom in **ooc** with a very simple implementation:

1. In your class's implementation source file define an external function prototype, like this:

   ```
   extern void injectMyClass( MyClass self );
   ```

   This is your injector method for your class.
2. In your class's constructor call your injector. It is the injector's responsibility to fill the members with the dependencies of your class. Do not set those members in your constructor. It is a good idea to put some check in your code that validates the injected dependencies, at least in the debug version.

   ```
   Static
   Void
   MyClass_constructor( MyClass self, void * param )
   {
      chain_constructor( MyClass, self, NULL );
      injectMyClass( self );
      assert( ooc_isInstanceOf( self->settings, Settings ) );
   }
   ```
3. Create your injector method in a separate file, let's call it `injector.c`.

   ```
   #include "myclass.h"
   #include "implement/myclass.h"
   #include "settings.h"

   void injectMyClass( MyClass self )
   {
      self->settings = settings_new();
   }
   ```

   You can inject singleton objects too, if you create them before any call to injectors, or you can use reference counted objects for your dependencies alternatively.
   Implement your other injectors in this file too.
4. Compile and link together your object files.
5. Copy your injector file into your unit testing directory, and rewrite it as it is required by your unit tests. Compile and link your test: pay atention to link your test injector, instead of the original one! (The supplied Makefiles will take care.)
   See [Using fake objects], page 42.

Altough in Java there is only a idiom called "inject while constructing", in **ooc** it makes sense "inject while initializing". Implement injectors for your initialization methods separately, when it makes sense. (e.g. Injecting virtual functions, like mixins into the virtual table.)

Important! Take care of your design in your classes, that if there is a circular dependency (typically a reference-back) between your classes, this approach will result in an endless loop! Break up the circular dependency, and pass the back reference as a constructor parameter!

9 Class manipulation tool

Creating **ooc** classes by typing from scratch may be labor-intensive, error prone, but mostly boring. Fortunatley **ooc** has a tool that helps you create classes from templates, or from other classes that are already implemented. This tool is suprisingly called *ooc* and is used as follows. Type at the prompt:

```
~$ ooc --new MyClass
```

This instruction creates a class called MyClass from the default ooc template, and puts it into the current working directory. Using the default template the following files will be created in the current working directory:

myclass.h
: This is the MyClass header containing the declaration of MyClass class plus its virtual functions. These are the publicly available declarations for MyClass, this class is to be included by the users of the class. Extend this file with your method declarations as needed.

myclass.c
: This is the class implementation file. It contains the class allocation, constructor, descructor etc. skeletons. You must extend this file with your method definitions, and other code.

implement/myclass.h
: This is the implementation header. This cointains the declaration of class data members, that are publicly not available. This file must be included by the subclasses of MyClass. (If you create your classes with the *ooc* tool with *--from* or *--source* switches then these includes are handled automatically.)

As you have created your class using the *ooc* tool, check the created skeletons, and modify them as needed. The created class can not be compiled without some modifications, and this is intentional: this forces you to set the construction parameters properly for example.

The following switches can be used with *ooc*. The switches can be combined!

ooc commands:

`--help` Prints the version information and a short help message.

`--new ClassName`
: Creates a new class named as ClassName.

`--testsuite`
: Creates a testsuite from the current directory. (Copies all makefiles and testsuite script/batch, that is required for unit testing.)

Modifiers for the *ooc --new ClassName* command:

`--parent ParentClassName`
: The created class is created as a subclass for ParentClassName. If `--parent` switch is missing, then `Base` will be used as parent class.

`--from SampleClassName`
: Uses SampleClassName as a template. If `--from` switch is missing, then the default `SampleClass` template is used.
If `--source` switch is not defined then the template class is searched for in the files called `sampleclassname.c`, `sampleclassname.h` and `implement/sampleclassname.h` in the default template directory (usually `/usr/local/share/ooc/template`).

`--source filename`
`--template filename`
: Uses `filename.c`, `filename.h` and `implement/filename.h` files as the template file.
If `filename` is a simple filename (not absolute path) then it is located in the standard template directory (usually `/usr/local/share/ooc/template`). If `--from` switch is not defined then the default `SampleClass` is looked for as a template in `sampleclassname`.

`--target filename`
: Puts the results into `filename.c`, `filename.h` and `implement/filename.h` files in the current working directory. (Depending on the template used, some files may not be used.)
Does not overwrite the files, appends the new content to the end of the files.

The next example creates a Foo class with header and implementation files, and adds FooException to the files as an additional class.

```
~$ ooc --new Foo --parent Bar
~$ ooc --new FooException --parent Exception --template private --target foo
```

You will have `foo.c`, `foo.h` and `implement/foo.h` files with class definitions and implementations for `Foo` and `FooException`.

The following templates are available:

`sample`
: The default template. This generates a class with 'protected' data members and implementation header. This class can be subclassed. (This is the default if you do not specify a *`--template`* or *`--source`*.)

`private`
: Class definition with private members only. This type of class can not be subclassed! (there is no implementation header)

`minimal`
: A minimal class definition, only with a class implementation file (no headers). Use this for internal classes within a class.

`interface`
: Interface declaration.

`mixin`
: Mixin declaration and implementation files.

`test`
: Unit testing base class template.

Appendix A GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and

that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called

an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Table of Figures

Index

(Index is nonexistent)