Classes in C

This document describes the simplest possible coding style for making classes in C. It will describe constructors, instance variables, instance methods, class variables, class methods, inheritance, polymorphism, namespaces with aliasing and put it all together in an example project.

1. C Classes

A ***class*** consists of an instance type and a class object:

An ***instance type*** is a `struct` containing variable members called ***instance variables*** and function members called instance methods. A variable of the instance type is called an ***instance***.

A ***class object*** is a global `const struct` variable containing ***class variables*** and class methods. These members belong the whole class without any references to any instances.

A class named "Complex" should name the instance type `struct Complex` and the class object `Complex`, and put the interface definitions in "Complex.h" and the implementation in "Complex.c".

```
Complex.h:
        struct Complex {
                ...
        };
        extern const struct ComplexClass {
                ...
        } Complex;

Complex.c:
        #include "Complex.h"
        const struct ComplexClass Complex={...};
```

+Rationale

2. Constructors

Instances must be initialized by ***constructors*** when declared, and the constructors must be class methods. The constructor should preferably return an instance type, but may also return a pointer to an instance type.

The Complex class gets two instance variables `re` and `im`, and a constructor named `new`:

```
Complex.h:
        struct Complex {
                double re, im;
        };
        extern const struct ComplexClass {
                struct Complex (*new)(double real, double imag);
        } Complex;

Complex.c:
        #include "Complex.h"
        static struct Complex new(double real, double imag) {
                return (struct Complex){.re=real, .im=imag};
        }
        const struct ComplexClass Complex={.new=&new};

Complex_test.c:
        ...
        struct Complex c=Complex.new(3., -4.);
```

3. Methods

Instance methods must be declared as instance type members *pointing to* the wanted function prototype, and that pointer must be set by the constructor. Typically, the method pointer is set to a `static` function defined in the implementation file.

To be able to access the instance's data, instance methods must receive a pointer to the instance as its first argument. This argument is typically named `this`.

When we add an instance method `abs()` for calculating the absolute value of a complex number we get:

```
Complex.h:
        struct Complex {
                double re, im;
                double (*abs)(struct Complex *this);
        };
        extern const struct ComplexClass {
                struct Complex (*new)(double real, double imag);
        } Complex;

Complex.c:
        #include "Complex.h"
        static double abs(struct Complex *this) {
                return sqrt(this->re*this->re+this->im*this->im);
        }
        static struct Complex new(double real, double imag) {
                return (struct Complex){.re=real, .im=imag, .abs=&abs};
        }
        const struct ComplexClass Complex={.new=&new};

Complex_test.c:
        ...
        struct Complex c=Complex.new(3., -4.);
        printf("%g\n", c.abs(&c)); // Prints 5
```

Class methods must be initialized in the same way as instance methods, but have no restriction on the prototype.

4. Inheritance

A ***base class*** must be represented as a member variable with the same name and type as the base class itself.

A subclass may ***override*** the base class instance method pointers to provide ***polymorphism***. The subclass must override with an identically prototyped function and set the base class' method pointer in the constructor *after* the baseclass' constructor has been called.

Whenever an overriden instance method is called, we are *guaranteed* that it was called by an instance of the *baseclass*. Since the instance method receives a pointer to the base class as its first argument, we may get the subclass using the `offsetof()` macro from `stddef.h`.

The following files shows a simple example of inheritance and polymorphism:

+Employee.h

```
struct Employee {
        const char *first_name;
        const char *family_name;
        const char *(*print)(struct Employee *this,
                             size_t bufsize,char buf[bufsize]);
};

extern const struct EmployeeClass {
        struct Employee (*new)(const char *first_name,
                               const char *family_name);
} Employee;
```

+Employee.c

+Manager.h

```
#include "Employee.h"

struct Manager {
        struct Employee Employee;
        int level;
};

extern const struct ManagerClass {
        struct Manager (*new)(const char *first_name,
                              const char *family_name, int level);
} Manager;
```

+Manager.c

+inheritance.c

```
#include "Manager.h"

int main(void)
{
        struct Manager manager=Manager.new("Håkon", "Hallingstad", 3);
```

```
		struct Employee employee=Employee.new("Håkon", "Hallingstad");
		struct Employee *polymorph=&manager.Employee;
		char buf[50];
		printf("%s\n", employee.print(&employee, sizeof(buf), buf));
		printf("%s\n", polymorph->print(polymorph, sizeof(buf), buf));
		return 0;
	}
```

The Manager class overrides Employee's print() instance method with the line from Manager.c:

```
	ret.Employee.print=&print;
```

Which makes inheritance.c print:

```
	Name: Håkon Hallingstad
	Name: Håkon Hallingstad, level 3
```

5. Controlling Access to Members

In object oriented languages each member has an access attribute and the compiler will enforce that access attribute.

With Classes in C we should use comments to specify the access attributes. For instance we use the following notation:

```
	struct Complex {
		...
		// protected:
		...
		// private:
		...
	};
```

6. Abstract Classes, Abstract Methods and Interfaces

In object oriented languages we can specify an abstract class to guarantee that the class cannot be instanciated. Abstract methods and interfaces can be used to guarantee that subclasses override methods.

With Classes in C just have to make sure any user of the class understands such intensions, for instance:

```
	struct ElementInterface {
		...
	};

	/*interface*/ struct Element {
		...
	};

	/*abstract*/ struct Complex {
		...
	};

	struct Stack {
		/*abstract*/ double (*foo)(struct Stack *this);
	};
```

Abstract instance method pointers should be initialized to `NULL`.

7. Namespaces

A ***namespace*** defines a common prefix of all identifiers exported by a class and the path of its header- and implementation- files.

For instance a `Complex` class with a namespace `org_pvv_hakonhal_utils_Complex` should have its implementation file in `org/pvv/hakonhal/utils/Complex.c`, and its header file in `org/pvv/hakonhal/utils/Complex.h` containing:

```
#ifndef ORG_PVV_HAKONHAL_UTILS_COMPLEX_H
#define ORG_PVV_HAKONHAL_UTILS_COMPLEX_H

struct org_pvv_hakonhal_utils_Complex {
        ...
};

extern struct org_pvv_hakonhal_utils_ComplexClass {
        ...
} org_pvv_hakonhal_utils_Complex;

#endif
```

When we are going to use the class we may ***alias*** the identifiers to make them more managable, by using the `#define` directive:

```
#include "org/pvv/hakonhal/utils/Complex.h"
#define Complex org_pvv_hakonhal_utils_Complex
...
struct Complex c=Complex.new();
```

8. An Example Project

In this example project we will create and test a bounds-checking stack implementation by extending a simpler stack implementation. The project will illustrate everything about C Classes including constructors, methods, inheritance, namespaces and aliases.

For compiling this project you should read C Project Building.

The Libray Project

We imagine the simple stack project has been downloaded from the net and the header file may be referenced as `org/somewhere/someone/Stack.h`. The stack header contains:

+Stack.h

```
#ifndef ORG_SOMEWHERE_SOMEONE_STACK_H
#define ORG_SOMEWHERE_SOMEONE_STACK_H

struct org_somewhere_someone_Stack_ElementI {
};

#define ORG_SOMEWHERE_SOMEONE_STACK_SIZE 100

struct org_somewhere_someone_Stack {
```

```
        void (*push)
        (struct org_somewhere_someone_Stack *this,
         struct org_somewhere_someone_Stack_ElementI *element);

        struct org_somewhere_someone_Stack_ElementI *(*pop)
        (struct org_somewhere_someone_Stack *this);

        // protected:

        int count;

        struct org_somewhere_someone_Stack_ElementI *
        data[ORG_SOMEWHERE_SOMEONE_STACK_SIZE];
};

extern struct org_somewhere_someone_StackClass {
        struct org_somewhere_someone_Stack (*new)(void);
} org_somewhere_someone_Stack;

#endif
```

`org_somewhere_someone_Stack_ElementI` is an interface for the elements that is stored in the stack, and impose no restrictions on the elements stored since the `struct` has an empty body.

We choose a unique org_pvv_hakonhal_utils namespace and creates the `org/pvv/hakonhal/utils` directory where we will put our `BStack.c` and `BStack.h` files.

The only thing our class will do in addition to `org_somewhere_someone_Stack`, is to check the bounds when `push()`'ing and `pop()`'ing, so our instance type only holds the reference to the base class:

+BStack.h

```
#ifndef ORG_PVV_HAKONHAL_UTILS_BSTACK_H
#define ORG_PVV_HAKONHAL_UTILS_BSTACK_H

#include "org/somewhere/someone/Stack.h"

struct org_pvv_hakonhal_utils_BStack {

        struct org_somewhere_someone_Stack org_somewhere_someone_Stack;

};

extern const struct org_pvv_hakonhal_utils_BStackClass {
        struct org_pvv_hakonhal_utils_BStack (*new)(void);
} org_pvv_hakonhal_utils_BStack;

#endif
```

The implementation file is somewhat more complex.

+BStack.c

```
#include "org/pvv/hakonhal/utils/BStack.h"
#define BStack      org_pvv_hakonhal_utils_BStack
#define BStackClass org_pvv_hakonhal_utils_BStackClass

#include "org/somewhere/someone/Stack.h"
#define ElementI    org_somewhere_someone_Stack_ElementI
#define Stack       org_somewhere_someone_Stack
#define STACK_SIZE  ORG_SOMEWHERE_SOMEONE_STACK_SIZE
```

```
8       #include <stdio.h>
9       #include <stdlib.h>

10      static void (*base_push)(struct Stack *this, struct ElementI *element);

11      static void push(struct Stack *base, struct ElementI *element)
12      {
13              if (base->count>=STACK_SIZE) {
14                      fprintf(stderr, "%s", "Stack overflow!\n");
15                      exit(1);
16              }
17
18              base_push(base, element);
19      }

20      static struct ElementI *(*base_pop)(struct Stack *this);

21      static struct ElementI *pop(struct Stack *base)
22      {
23              if (base->count<=0) {
24                      fprintf(stderr, "%s", "Stack underflow!\n");
25                      exit(1);
26              }
27              return base_pop(base);
28      }

29      static struct BStack new(void)
30      {
31              struct BStack ret;
32              ret.Stack=Stack.new();

33              base_push=ret.Stack.push;
34              ret.Stack.push=&push;

35              base_pop=ret.Stack.pop;
36              ret.Stack.pop=&pop;

37              return ret;
38      }

39      const struct BStackClass BStack={.new=&new};
```

Let us look at the implementation of the `push()` instance method, the `pop()` is similar. Since it is using the base class' `push()` instance method, we must keep a pointer to the base class' instance method, see 10, 18 and 33-34.

Testing the Library Project

The implementation of the `Bstack` class is rather long:

+BStack_test.c

```
#include "org/pvv/hakonhal/utils/BStack.h"
#define BStack org_pvv_hakonhal_utils_BStack

#include "org/somewhere/someone/Stack.h"
#define ElementI org_somewhere_someone_Stack_ElementI
#define Stack org_somewhere_someone_Stack

#include <stddef.h>
#include <stdio.h>

struct Integer {
        struct ElementI ElementI;
        int value;
        void (*print)(struct Integer *this, const char *id);
};

static void print(struct Integer *this, const char *id)
```

```
14      {
15              printf("%s: %d\n", id, this->value);
16      }

17      static struct Integer new(int value)
18      {
19              return (struct Integer){
20                      .ElementI={},
21                      .value=value,
22                      .print=&print,
23              };
24      }

25      static const struct {
26              struct Integer (*new)(int value);
27      } Integer={.new=&new};

28      int main(void)
29      {
30              struct BStack stack=BStack.new();
31              struct Integer i=Integer.new(10), j=Integer.new(20);
32              struct Integer *ptr;

33              stack.Stack.push(&stack.Stack, &i.ElementI);
34              stack.Stack.push(&stack.Stack, &j.ElementI);

35              ptr=(void *)stack.Stack.pop(&stack.Stack)-offsetof(struct Integer, ElementI);
36              ptr->print(ptr, "j");
37              ptr=((void *)stack.Stack.pop(&stack.Stack))-offsetof(struct Integer, ElementI);
38              ptr->print(ptr, "i");
39              printf("%s\n", "Will now try to pop an empty stack");
40              stack.Stack.pop(&stack.Stack);

41              return 0;
42      }
```

We define an `Integer` class that extends the `org_somewhere_someone_Stack_ElementI` interface so that it might be added to our `BStack`, see lines 8-27. The class also contains an int and an instance method to print it. Note that since we are constructing an executable, the `Integer` class does not need to have a namespace.

Since we are using aliasing, statements such as that on line 30 actually reads:

```
struct org_pvv_hakonhal_utils_BStack stack=org_pvv_hakonhal_utils_BStack.new();
```

Since the `push()` and `pop()` instance methods where defined by `Stack`, we need to "go through" the `Stack` subclass to call them, as seen on e.g. line 33-34.

When we retreive the elements on the stack, we need the slightly awkward syntax in 35 and 37. Conceptually, we receive a pointer to the `ElementI` base class of a `struct BStack` variable, so we just need to shift it.

The `BStack_test` executable will output:

```
j: 20
i: 10
Will now try to pop an empty stack
Stack underflow detected!
```

Håkon Hallingstad hakonhal@pvv.org

Last modified: Mon Jun 21 09:00:46 CEST 2004