

<< *Draft* >>

Useful Polymorphism in C language

Gabriel Gonzalez Garcia
Software Engineer @ Advansen
gabriel [] advansen (dot) com

June 22, 2006

Abstract

Several documents have been written showing how to implement Object-Oriented flavored programming in preferred language *C*. Despite of those great tricky, well-thought and potentially useful features, they fail in delivering easy programming syntax since they usually rely in complex definitions and non-trivial pointer use. In this paper I present a new approach on adding one of the most useful object oriented techniques, the well-known *Polymorphism*, just using it in a symplified way without obfuscating the clear and traditional C syntax but keeping its powerfulness.

1 Introduction

After two decades of evolution of Object Oriented theory and practice no one can neglect how useful is map “real” world objects to application domain solution. Not only it makes easier to design a better blueprint for developers but also it helps last ones to understand what to code and therefore make application less error prone¹ and easier to mantain.

Aside all of these helpful features most of the community of C users still regret to apply them into their design loosing their benefits. Maybe because there is not a straightforward method to make C OO-flavored without fighting with complex syntax delivered via *#define*

In this paper I present how one can achive it and help his/her project just doing things in the clearest way i.e. using C structures and pointers with a good distribution of headers and a good function labeling.

2 Structure Overlapping in C

Structure Overlapping² is the basis of the object oriented techniques in C, since it allows to cast between diffent types without loosing any information. Let me define two types:

¹Accurated to requirements, not less error prone to language specific flaws.

²I do now know wheter this term has been used before with the same meaning.

```

struct parent
{
    int  parent_id;
    char *name;
};

```

```

struct child
{
    int  child_id;
    char *name;
};

```

Maybe you have guessed that the name of the types is related in same way to the inheritance relationships in the object oriented paradigm. You are right but let me define another two types:

```

typedef struct parent* Parent;
typedef struct child* Child;

```

Using this new two type we improve the readability of our code³ since they are shorter and let you concentrate in the meaning of the data rather than the structure. Let me go an step further:

```

typedef struct parent
{
    int  parent_id;
    char *name;
} Parent_t;
typedef Parent_t* Parent;

```

```

typedef struct child
{
    Parent_t  parent;
    int      child_id;
    char     *name;
} Child_t;
typedef Child_t* Child;

```

Now we have another two new types `Parent_t` and `Child_t`, they are helpful when declaring a non-pointer type and when you want the size of the structure. I have embedded a `Parent` structure in the `child` as well. Why could this be useful? A bunch of code is better than a thousand words:

```

{
    Parent  *parent;
    Child  *child;

    child = (Child) calloc(1, sizeof(Child_t));

    child->child_id = 1;
}

```

³You can remove the pointer from the typedef if it helps you defining a variable with `*` to identify them as a pointer.

```

child->parent.parent_id = 2;

*
* Here is when the structure overlapping come in play
*
parent = (Parent) child;
parent->parent_id = 3;
}

```

I hope you have realized the usefulness of the overlapping structures and the way it is related to the object oriented approach. The point is you can have common information you can share between different modules. In the next section I present a simple example which exploits this powerful feature of C.

3 Polimorphism in C

As I have noted in previous sections by implementing polymorphism in C I have kept the semantic avoiding the syntax of object oriented languages this leads in clear, powerful and mantenible code which will help us in developing large projects rather than obfuscating them. To illustrate this proposal I will be using the typical example of two data: types a single list and a double-linked list. In this section I present only the code needed to highlight the main ideas behind the article, the full source code can be found in the appendix.

Since we are using an object oriented feature we can use UML to draw the a class diagram which classes will be mapped as C modules.

3.1 Organizing the source

Following the approach of [1] the best place and names for the header files is the shown below, being our main module called `data_types`:

```

include/data_types
include/data_types/dt_general.h
include/data_types/dt_general_defs.h
include/data_types/dt_general_iface.h
include/data_types/concrete_types
include/data_types/concrete_types/dt_single_list.h
include/data_types/concrete_types/dt_single_list_defs.h
include/data_types/concrete_types/dt_single_list_iface.h
include/data_types/concrete_types/dt_double_linked_list.h
include/data_types/concrete_types/dt_double_linked_list_defs.h
include/data_types/concrete_types/dt_double_linked_list_iface.h

```

The `_iface.h` suffix includes the functions exported from the module and `_defs.h` definitions and new types.

Derived from the headers order we choose the next places for the source code:

```

src/data_types
src/data_types/dt_general.c
src/data_types/concrete_types/single_list

```

```

src/data_types/concrete_types/single_list/single_list.c
src/data_types/concrete_types/single_list/single_list.h
src/data_types/concrete_types/double_linked_list/double_linked_list.c
src/data_types/concrete_types/double_linked_list/double_linked_list.h

```

3.2 Modules's definitions

Understanding how overlapping structures works is the key idea of getting the whole view of the polymorphism in C. Below I show the structures for each module and how they are coupled to each other. The key point is understand how and when the pointer to functions should be called, anyway I will explain it later.

```

$ cat dt_general_defs.h
...
struct _data_type;
typedef struct _data_type DataType_t;
typedef DataType_t* DataType;
struct _data_type
{
    char    *data_type_id;

    boolean (*insert_item)(DataType, void *);
    boolean (*delete_item)(DataType, void *);
    boolean (*destroy)(DataType, void (*)(void *));

    boolean (*next_item)(DataType, void (*)(void *));
    void    *(*get_current)(DataType);

    int     no_items;
};

```

These pointer will store the address of the specific functions of the concrete data type which has been instantiated, i.e. if we have created a new single list these functions will be those exported from the single_list module.

```

$ cat dt_single_list_defs.h
...
struct _dt_single_list;
typedef struct _dt_single_list DTSingleList_t;
typedef DTSingleList_t* DTSingleList;
struct _dt_single_list
{
    DataType_t parent;

    void    *head;
    void    *current;
};
...
$ cat dt_single_list_defs.h
...

```

```

struct _dt_double_linked_list;
typedef struct _dt_double_linked_list DTDoubleLinkedList_t;
typedef DTDoubleLinkedList_t* DTDoubleLinkedList;
struct _dt_double_linked_list
{
    DataType_t parent;

    void      *head;
    void      *current;
};
...

```

3.3 Module's interfaces

Here I show the interfaces implemented for each module beginning with the general module `data_type`, the common functionalities to all concrete data types:

```

$ cat dt_general.c
...
boolean dt_insert_item(DataType, void *);
boolean dt_delete_item(DataType, void *, void (*)(void *));
boolean dt_destroy(DataType, void (*)(void *));

boolean dt_next_item(DataType);
void dt_get_current(DataType);

#define dt_is_empty(datatype) (!datatype->no_items)
...

```

Below, the concrete module functions are shown, as you can notice there is an extra create function here. Why do not have our general data type module one as well? Because it is an abstract module it can be instantiated, you can not store anything if you do not know how.

```

$ cat concrete_types/dt_single_list.c
...
DataType *dt_single_list_create();
boolean dt_single_list_insert_item(DataType, void *);
boolean dt_single_list_delete_item(DataType, void *, void (*)(void *));
boolean dt_single_list_destroy(DataType, void (*)(void *));

boolean dt_single_list_next_item(DataType);
void dt_single_list_get_current(DataType);
...

$ cat concrete_types/dt_double_linked_list.c
...
DataType *dt_double_linked_list_create();
boolean dt_double_linked_list_insert_item(DataType, void *);
boolean dt_double_linked_list_delete_item(DataType, void *, void (*)(void *));

```

```

boolean    dt_double_linked_list_destroy(DataType, void (*)(void *));
boolean    dt_double_linked_list_next_item(DataType);
void       *dt_double_linked_list_get_current(DataType);
...

```

3.4 Getting all working

A brief example of how we can get all the stuff developed in the previous sections.

```

{
    DataType list;
    struct my_data
    {
        int a;
        char c;
    } *item;

    item = (struct my_data *) calloc(1, sizeof(struct my_data));

    /*
     * We choose the concrete data type
     */
    list = dt_single_list_create();

    /*
     * And now we can use it in a abstract way, without regarding
     * which is the underlying structure.
     */
    if (!dt_insert_item(list, item))
    {
        /* error handling */
    }

    if (!dt_delete_item(list, item, free))
    {
        /* error handling */
    }

    ...
}

```

A Source Code