How to wrap C functions to OCaml

This document is a small tutorial that introduces how to call C functions from OCaml. Which is usefull if you want to make a mixed C and OCaml application, if you want to call some functions you need from a C library, or to write a complete binding to a C library.

Contents

Hello World

The two files "hello.ml" and "hello_stubs.c":

```
external print_hello: unit -> unit = "caml_print_hello"

let () =
  print_hello ()
```

```
#include <stdio.h>
#include <caml/mlvalues.h>

CAMLprim value
caml_print_hello(value unit)
{
    printf("Hello\n");
    return Val_unit;
}
```

Compile and run:

```
$ ocamlopt -o hello.opt hello.ml hello_stubs.c
$ ./hello.opt
Hello
```

Quick start

Let's start with the simpler case one can find, calling a C function with an integer parameter. This is achieved in an .ml file by this line:

```
external send_an_int: int -> unit = "get_an_int"
```

The external keyword tells the compiler that the function comes from outside the OCaml world. The compiler needs the function name which will be seen from the OCaml area, its type, and the name of the C function which will be called.
In the C source file, define a C function whose name is that given in the OCaml side between quotes. This function must have return type `CAMLprim value` and its parameter type has to be `value`. All parameters given to C functions from OCaml are of type `value`. The include header `<caml/mlvalues.h>` provides conversion macros to convert the type `value` to C native types. In this case to convert to a C integer, that macro is `Int_val()`. Then as the return value of the OCaml function is `unit`, the C function have to return a unit, which is done with the macro `Val_unit`.

```
#include <stdio.h>
#include <caml/mlvalues.h>

CAMLprim value
get_an_int( value v )
{
    int i;
    i = Int_val(v);
    printf("%d\n", i);
    return Val_unit;
}
```

Of course, think to include the C headers you need, the standard library or other libraries.

Compiling C and OCaml together

Then to compile these two files together, here are the command lines you will need:

```
ocamlc -i funs.ml > funs.mli
ocamlc -c funs.mli
ocamlc -c funs.ml
ocamlc -c wrap.c
ocamlmklib  -o  wrap_stubs  wrap.o
ocamlc -a  -custom  -o funs.cma  funs.cmo  -dllib dllwrap_stubs.so
```

But it will be more handy with a basic `Makefile`:

```
wrap.o: wrap.c
        ocamlc -c $<

dllwrap_stubs.so: wrap.o
        ocamlmklib  -o  wrap_stubs  $<

funs.mli: funs.ml
        ocamlc -i $< > $@

funs.cmi: funs.mli
        ocamlc -c $<

funs.cmo: funs.ml funs.cmi
        ocamlc -c $<

funs.cma:  funs.cmo  dllwrap_stubs.so
        ocamlc -a  -o $@  $<  -dllib -lwrap_stubs

funs.cmx: funs.ml funs.cmi
        ocamlopt -c $<

funs.cmxa:  funs.cmx  dllwrap_stubs.so
        ocamlopt -a  -o $@  $<  -cclib -lwrap_stubs

clean:
        rm -f *.[oa] *.so *.cm[ixoa] *.cmxa
```

And if you use a library in the file wrap.c, you need to add some extra compilation instructions to link against the library, which is described in the section **Linking against a library**.
Just a detail about the command `ocamlc -c wrap.c`, you can replace this line by one of the line below for cases when you wish to give extra arguments to gcc:

```
ocamlc -c -cc "gcc -o wrap.o" wrap.c
gcc -c -I"`ocamlc -where`" wrap.c
```

Also I would recommand trying to replace the command `ocamlc -c wrap.c` by `ocamlc -verbose -c wrap.c` so that the ocamlc compiler will print out all the internal commands that it will proceed.
Then you can test the result opening the .cma file in the top-level:

```
% ocaml funs.cma

# open Funs;;
# send_an_int 9 ;;
9
- : unit = ()
```

Or at your option from a test script:

```
#load "funs.cma" ;;
open Funs ;;

let () =
  send_an_int 5;
;;
```

You also have the option, if you just want to build an C-OCaml mixed application without to build an intermediate .cma / .cmxa binding, to directly compile the objects together:

```
make wrap.o
make funs.cmx
ocamlopt wrap.o funs.cmx -o myapp
./myapp
```

OK, Now if everything works, let's see how to convert the other types. floating point numbers, and strings:

```
external send_a_float: float -> unit = "get_a_float"
external send_a_string: string -> unit = "get_a_string"

CAMLprim value
get_a_float( value v )
{
    float f;
    f = Double_val(v);
    printf("%f\n", f);
    return Val_unit;
}

CAMLprim value
get_a_string( value v )
{
    char *s;
    s = String_val(v);
    printf("%s\n", s);
    return Val_unit;
}
```

The `Double_val()` conversion macro can be used both for C floats and C doubles.

Returning Basic Types

Now let's see how to return these basic types from C to OCaml.

```
external get_an_int: unit -> int = "send_an_int"

CAMLprim value
send_an_int( value unit )
{
    int i;
    i = 6;
    return Val_int(i);
}
```

The conversion macro `Val_int()` is similar to the macros seen before, it converts a C integer into an OCaml integer value.
For cases where C code uses long integers, instead of `Int_val()` just use `Long_val()`.
In the same way you can convert an OCaml boolean to a C integer with `Bool_val()`, and a C integer to an OCaml boolean with `Val_bool()`. For convenience there are also macros `Val_true` and `Val_false`.

Allocated Types

It goes a bit different for floats and strings than ints, since these types have to be allocated for OCaml, and one thing important to know in OCaml is that when an allocation is done (a new value created), a garbage collection may be done. If the allocation is given at the end point in the C `return` statement, all is fine. But notice that in other cases which will be seen below, additional statements have to be used.

```
external get_a_float: unit -> float = "send_a_float"
external get_a_string: unit -> string = "send_a_string"

#include <caml/alloc.h>
```

```
CAMLprim value
send_a_float( value unit )
{
    double d;
    d = 2.6;
    return caml_copy_double(d);
}

CAMLprim value
send_a_string( value unit )
{
    char *s = "the walking camel";
    return caml_copy_string(s);
}
```

In more complex functions where input values are still used after a new OCaml value is allocated, there is a risk that the input value cannot be used because it has been garbage collected when the new OCaml value is allocated. In such cases you have to use the macros `CAMLparamN()` (where *N* is the number of parameters) at the beginning of the function and at the end instead of `return` use `CAMLreturn()`. And for local allocated values, use the declaration `CAMLlocalN()`. For example you could write the previous functions "send_a_float" and "send_a_string" in this way:

```
CAMLprim value
send_a_float( value unit )
{
    CAMLparam1(unit);
    CAMLlocal1(ml_f);
    double d;
    d = 2.6;
    ml_f = caml_copy_double(d);
    CAMLreturn(ml_f);
}

CAMLprim value
send_a_string( value unit )
{
    CAMLparam1(unit);
    CAMLlocal1(ml_s);
    char *s = "the walking camel";
    ml_s = caml_copy_string(s);
    CAMLreturn(ml_s);
}
```

When you are not sure if you need to use these macro or not, it's better to use these to stay in peace with the garbage collector, since using these macros won't cause problems, even if they are not needed. That's why you will see below cases where they are used while not strictly needed. Moreover, a lot of OCaml users recommend always using these to avoid the risk of forgetting to use them when they are needed.
Also when using this set of macros, add this header where they are defined:

```
#include <caml/memory.h>
```

Constructed Types

Now let's see how to inspect and construct more complex structures like tuples, arrays, lists and records.

Accessing Tuples

To access the content of an OCaml tuple, the `Field()` macro is provided to access each piece of the tuple. Then each field is as usual of type value which needs to be converted to native C types.

```
external inspect_tuple: int * float * string -> unit = "inspect_tuple"

CAMLprim value
inspect_tuple( value ml_tuple )
{
    CAMLparam1( ml_tuple );
    CAMLlocal3( vi, vf, vs );

    vi = Field(ml_tuple, 0);
    vf = Field(ml_tuple, 1);
    vs = Field(ml_tuple, 2);

    printf("%d\n", Int_val(vi));
    printf("%f\n", Double_val(vf));
    printf("%s\n", String_val(vs));

    CAMLreturn( Val_unit );
}
```

You can check the result is as expected from the top-level:

```
# inspect_tuple (3, 2.4, "functional") ;;
3
2.400000
functional
```

```
- : unit = ()
```

Or the C part may have been written in a more concise way to get the same result:

```
CAMLprim value
inspect_tuple( value ml_tuple )
{
    printf("%d\n", Int_val(Field(ml_tuple,0)));
    printf("%f\n", Double_val(Field(ml_tuple,1)));
    printf("%s\n", String_val(Field(ml_tuple,2)));
    return Val_unit;
}
```

(Since there are no allocation here.)
Accessing fields of a record is exactly the same than for a tuple:

```
type rec_t = { a:int; b:float; c:string }
external inspect_record: rec_t -> unit = "inspect_record"

CAMLprim value
inspect_record( value ml_record )
{
    printf("%d\n", Int_val(Field(ml_record, 0)));
    printf("%g\n", Double_val(Field(ml_record, 1)));
    printf("%s\n", String_val(Field(ml_record, 2)));
    return Val_unit;
}
```

The index number of the fields are ordered as the fields appear in the OCaml declaration of the record type.

```
# inspect_record {a=26; b=4.3; c="camelids folks" } ;;
26
4.3
camelids folks
- : unit = ()
```

There is though a particular case when all the fields of the record are of type float, in this case the record is seen from C as a float array, see below for this particular case.

Accessing Arrays

Accessing fields of arrays is very similar:

```
external inspect_int_array: int array -> unit = "inspect_int_array"

CAMLprim value
inspect_int_array( value ml_array )
{
    CAMLparam1( ml_array );
    int i, len;
    len = Wosize_val(ml_array);
    for (i=0; i < len; i++)
    {
        printf("%d\n", Int_val(Field(ml_array, i)));
    }

    CAMLreturn( Val_unit );
}
```

The only trick here is to get the number of elements in the array. And also if the array is a `float array` the macros to get the number of elements and to access the content are a bit different:

```
external inspect_float_array: float array -> unit = "inspect_float_array"

CAMLprim value
inspect_float_array( value ml_float_array )
{
    CAMLparam1( ml_float_array );
    int i, len;
    len = Wosize_val(ml_float_array) / Double_wosize;
    for (i=0; i < len; i++)
    {
        printf("%g\n", Double_field(ml_float_array, i));
    }

    CAMLreturn( Val_unit );
}
```

If you test these two functions:

```
# inspect_int_array [| 2; 4; 3; 1; 9 |] ;;
2
4
3
1
9
- : unit = ()

# inspect_float_array [| 2.4; 5.8; 6.9; 12.2; 32.8 |] ;;
2.4
5.8
```

```
6.9
12.2
32.8
- : unit = ()
```

Accessing Lists

Now here is how you can access the content of an OCaml list:

```
external inspect_list: string list -> unit = "inspect_list"

CAMLprim value
inspect_list( value ml_list )
{
    CAMLparam1( ml_list );
    CAMLlocal1( head );

    while ( ml_list != Val_emptylist )
    {
        head = Field(ml_list, 0);  /* accessing the head */
        printf("%s\n", String_val(head));
        ml_list = Field(ml_list, 1);  /* point to the tail for next loop */
    }

    CAMLreturn( Val_unit );
}
```

Test it from the top-level:

```
# inspect_list ["hello"; "you"; "world"; "camelids"] ;;
hello
you
world
camelids
- : unit = ()
```

As you can see lists are constructed as pair of items, the first item in the pair is the head, and the second is the tail, which can be another list or the empty list which is written as `[]` on the OCaml side, and `Val_emptylist` on the C side.
As an illustration of this you can test from the top-level typing these commands:

```
# external trans: (int * (int * (int * (int * int)))) -> int list = "%identity" ;;
external trans : int * (int * (int * (int * int))) -> int list = "%identity"

# trans (1, (2, (3, (4, 0)))) ;;
- : int list = [1; 2; 3; 4]
```

Here "%identity" is a built-in OCaml function which just allows to convert a type to another, keeping its internal representation identical.
And you can also notice here that the last integer given is 0, since as you can see reading the `<caml/mlvalues.h>` header that `Val_emptylist` is defined as `Val_int(0)`.

Creating Tuples and Arrays

Now let's see how to construct a tuple on the C side and returning it to OCaml:

```
external create_tuple: 'a -> 'b -> 'c -> 'a * 'b * 'c = "create_tuple"

CAMLprim value
create_tuple( value a, value b, value c )
{
    CAMLparam3( a, b, c );
    CAMLlocal1( abc );

    abc = caml_alloc(3, 0);

    Store_field( abc, 0, a );
    Store_field( abc, 1, b );
    Store_field( abc, 2, c );

    CAMLreturn( abc );
}

# create_tuple 38 'G' "www.ifrc.org" ;;
- : int * char * string = (38, 'G', "www.ifrc.org")
```

The first parameter of `caml_alloc()` is the number of fields, and the second one indicates the tag of the value, here it is 0 because for ordinary ocaml values like tuples there is no tags needed.
And it works the same for records, as we have seen they have the same internal representation as tuples, except when the record contains only floats.
This also works for arrays except that you have to make sure that all the elements of the array have the same type. All arrays can be created this way except `float array`s that have a different internal representation.
In the same manner than the list, you can verify that records and tuples have the same internal representation with:

```
# type rec_b = { i:int; c:char; s:string } ;;
```

```
type rec_b = { i : int; c : char; s : string; }

# external trans: rec_b -> int * char * string = "%identity" ;;
external trans : rec_b -> int * char * string = "%identity"

# trans { i=38; c='G'; s="www.ifrc.org" } ;;
- : int * char * string = (38, 'G', "www.ifrc.org")

# external create_rec_b: int -> char -> string -> rec_b = "create_tuple" ;;
external create_rec_b : int -> char -> string -> rec_b = "create_tuple"

# create_rec_b 38 'G' "www.ifrc.org" ;;
- : rec_b = {i = 38; c = 'G'; s = "www.ifrc.org"}
```

Creating Lists

At this point you should be able to find how to build an OCaml list form C. Here is an exemple:

```
external string_explode: string -> char list = "create_list"

CAMLprim value create_list( value ml_str )
{
    CAMLparam1( ml_str );
    CAMLlocal2( cli, cons );

    char *str = String_val(ml_str);
    int len = caml_string_length(ml_str);
    int i;

    cli = Val_emptylist;

    for (i = len - 1; i >= 0; i--)
    {
        cons = caml_alloc(2, 0);

        Store_field( cons, 0, Val_int(str[i]) );  // head
        Store_field( cons, 1, cli );              // tail

        cli = cons;
    }

    CAMLreturn( cli );
}

# string_explode "OCaml" ;;
- : char list = ['O'; 'C'; 'a'; 'm'; 'l']
```

Creating Floats arrays

And as the floats arrays are particular, they are a bit different to create from C, notice both the length trick and the tag when allocating:
`float_array = caml_alloc(length * Double_wosize, Double_array_tag);`
And to store elements use `Store_double_field()` instead of `Store_field()`, and there's no need to use the `caml_copy_double` in it.

Enums / Variants

For wrapping C enums, or for wrapping params defined as constant with `#define` the method is the same. Both can be wrapped on OCaml enumerated variants with constants constructors without parameters. From the C site the constructors is represented as integers ordered from 0 to (N - 1) (where N is the number of constructors in the variant type). So basically you can just use a `switch` or an array containing all these values make match the C value. For instance:

```
type moving =
  | WALKING
  | RUNNING
  | SWIMMING
  | FLYING

external send_enum: moving -> unit = "wrapping_enum_ml2c"

typedef enum _moving {
    WALKING,
    RUNNING,
    SWIMMING,
    FLYING
} moving;

CAMLprim value
wrapping_enum_ml2c( value v )
{
    moving param;
    switch (Int_val(v)) {
        case 0: param = WALKING; break;
```

```
            case 1: param = RUNNING; break;
            case 2: param = SWIMMING; break;
            case 3: param = FLYING; break;
        }
        switch (param) {
            case WALKING:  puts("Walking"); break;
            case RUNNING:  puts("Running"); break;
            case SWIMMING: puts("Swimming"); break;
            case FLYING:   puts("Flying"); break;
        }
        return Val_unit;
    }
```

Here it is very important to make match the good number in the switch according to the place in the enumeration in the OCaml variant.
To send back an constant variant from C to OCaml, just return `Val_int()` with its corresponding index.
If there are a lots of possible values in the enumeration (or possible constants), you can also use an array in stead of a switch:

```
static const moving table_moving[] = {
    WALKING,
    RUNNING,
    SWIMMING,
    FLYING
};

CAMLprim value
wrapping_enum_ml2c( value v )
{
    moving param;
    param = table_moving[Long_val(v)];

    switch (param) {
        case WALKING:  puts("Walking"); break;
        case RUNNING:  puts("Running"); break;
        case SWIMMING: puts("Swimming"); break;
        case FLYING:   puts("Flying"); break;
    }
    return Val_unit;
}
```

Here you need to be sure not to try to access to an index in the association table after the last one. Which could occur if you add an item on the OCaml side, and forget to add it also in the array on the C side.
Something you could do is checking if the index requested is not greater than the last one, and if it is raising an exception.

Convert C bit fields

Imagine you have a C bit field defined as below in your header file:

```
#define ShiftMask      (1<<0)
#define LockMask       (1<<1)
#define ControlMask    (1<<2)
#define Mod1Mask       (1<<3)
```

to get it wrapped, the most common solution is to convert it to a variant list:

```
type state =
  | ShiftMask
  | LockMask
  | ControlMask
  | Mod1Mask

type state_mask = state list
```

As previously build a table with the C values in the same *exact* order than in the OCaml variant, and iter all items of the caml list to set the C bit field accordingly:

```
static const int state_mask_table[] = {
    ShiftMask,
    LockMask,
    ControlMask,
    Mod1Mask
};

static inline int
state_mask_val( value mask_list )
{
    int c_mask = 0;
    while ( mask_list != Val_emptylist )
    {
        value head = Field(mask_list, 0);
        c_mask |= state_mask_table[Long_val(head)];
        mask_list = Field(mask_list, 1);
    }
    return c_mask;
}
```

And to convert the C bit field to the OCaml variant list, you need to test the bit of every field, and if it matches push the associated OCaml value to the list:

```
#define Val_ShiftMask    Val_int(0)
#define Val_LockMask     Val_int(1)
#define Val_ControlMask  Val_int(2)
#define Val_Mod1Mask     Val_int(3)

static value
Val_state_mask( int c_mask )
{
    CAMLparam0();
    CAMLlocal2(li, cons);
    li = Val_emptylist;

    if (c_mask & ShiftMask) {
        cons = caml_alloc(2, 0);
        Store_field( cons, 0, Val_ShiftMask );
        Store_field( cons, 1, li );
        li = cons;
    }
    if (c_mask & LockMask) {
        cons = caml_alloc(2, 0);
        Store_field( cons, 0, Val_LockMask );
        Store_field( cons, 1, li );
        li = cons;
    }
    if (c_mask & ControlMask) {
        cons = caml_alloc(2, 0);
        Store_field( cons, 0, Val_ControlMask );
        Store_field( cons, 1, li );
        li = cons;
    }
    if (c_mask & Mod1Mask) {
        cons = caml_alloc(2, 0);
        Store_field( cons, 0, Val_Mod1Mask );
        Store_field( cons, 1, li );
        li = cons;
    }

    CAMLreturn(li);
}
```

Exchanging Raw Data

If you have to handle raw data, you can do so using OCaml strings wherever there are array of bytes or void* in C.
On the OCaml side, strings are able to handle any character, included the possible '\000', but if you use `caml_copy_string()` to copy the raw data it will end at the first Null character, because it considers it as the string terminator.
So you should use `caml_alloc_string()` and `memcpy()`:

```
#include <caml/mlvalues.h>
#include <caml/alloc.h>
#include <string.h>

CAMLprim value get_raw_data( value unit )
{
    CAMLlocal1( ml_data );
    char * raw_data;
    int data_len;
    /*
     * initialise raw_data and data_len here
     */
    ml_data = caml_alloc_string(data_len);
    memcpy( String_val(ml_data), raw_data, data_len );
    return ml_data;
}
```

Notice that here it is possible not to use CAMLparam/CAMLreturn even if there is an ocaml alloc, because there aren't any ocaml values used *after* the ocaml alloc. In your code if there are some, do put the CAMLparam/CAMLreturn !
If the data chunk represents something like a matrix (an image for example) you will probably prefer to use an OCaml bigarray which is a very handy data structure for this task because it shares the same memory layout than in C.

In your code, if the buffer `raw_data` is C mallocated and then filled by some process, it is even possible to only allocate an ocaml string and then get a pointer to the first byte of this ocaml string. So in such cases you will save one C malloc(), one C free(), and the memcpy() call.

```
CAMLprim value get_raw_data( value some_param )
{
    CAMLparam( some_param );
    CAMLlocal1( ml_data );
    char * raw_data;
    int data_len;
    /* do initialise data_len */
    ml_data = caml_alloc_string( data_len );
    raw_data = String_val(ml_data);
    /*
       Now you can use raw_data and fill it as if it was
       a buffer of type (char *) of length data_len.
       (once given to ocaml it will be garbage-collected as every ocaml value)
    */
    CAMLreturn( ml_data );
}
```

If you need to access to one byte of the buffer like this `raw_data[n]`, it is also possible to make the same access directly on the ocaml value like this `Byte(ml_data, n)`.
(And if raw_data was of type `(unsigned char *)` replace `Byte()` by `Byte_u()`.)

More than 5 parameters

There is a special case for OCaml functions that have more than 5 parameters, in such cases you have to provide 2 C functions, as you can see below:

```
external lots_of_params: p1:int -> p2:int -> p3:int ->
                         p4:int -> p5:int -> p6:int -> p7:int -> unit
    = "lotprm_bytecode"
      "lotprm_native"

CAMLprim value
lotprm_native( value p1, value p2, value p3,
               value p4, value p5, value p6, value p7 )
{
    printf("1(%d) 2(%d) 3(%d) 4(%d) 5(%d) 6(%d) 7(%d)\n",
           Int_val(p1), Int_val(p2), Int_val(p3),
           Int_val(p4), Int_val(p5), Int_val(p6), Int_val(p7) );
    return Val_unit;
}

CAMLprim value
lotprm_bytecode( value * argv, int argn )
{
    return lotprm_native( argv[0], argv[1], argv[2],
                          argv[3], argv[4], argv[5], argv[6] );
}
```

And as you can see the function that will be called in bytecode compiled programs have a different prototype. Then in the bytecode function, the common and easiest solution is to call the native one.
And another detail for those functions when you need to protect the values from the garbage collector, the macros CAMLparam*N*() exist only for *N* from 1 to 5. Then put the 5 first parameters in CAMLparam5(), and use additional CAMLxparam*N*() macros which exist too with *N* from 1 to 5. So for example if you have 13 parameters, use one CAMLparam5(), one CAMLxparam5() and one CAMLxparam3(). You have to use only one macro without the "x" in the middle, and you can use as many of the one with the "x".
Here is the same function with these macros:

```
CAMLprim value
lotprm_native( value p1, value p2, value p3,
               value p4, value p5, value p6, value p7 )
{
    CAMLparam5(p1, p2, p3, p4, p5);
    CAMLxparam2(p6, p7);

    printf("1(%d) 2(%d) 3(%d) 4(%d) 5(%d) 6(%d) 7(%d)\n",
           Int_val(p1), Int_val(p2), Int_val(p3),
           Int_val(p4), Int_val(p5), Int_val(p6), Int_val(p7) );

    CAMLreturn(Val_unit);
}

CAMLprim value
lotprm_bytecode( value * argv, int argn )
{
    return lotprm_native( argv[0], argv[1], argv[2],
                          argv[3], argv[4], argv[5], argv[6] );
}
```

Raising Exceptions

Raising the predefined exceptions of OCaml is very easy. For example for raising the `Invalid_argument` exception, the C function is:

```
    caml_invalid_argument("Error message");
```

And to raise an exception `Failure`:

```
    caml_failwith("Error message");
```

The header file to include when using these functions is:

```
#include <caml/fail.h>
```

And if your module defines custom exceptions, it is possible to raise these from C as explained on the page
http://caml.inria.fr/pub/docs/manual-ocaml/manual033.html
in section **19.7.3** .

Pointers to C structures

Often C libraries use types defined from structures or pointers to structures. OCaml can safely handle pointers to memory allocated areas. To achieve this task, the simpler method is to cast your pointer to and from the type `(value)` which can handle pointers.
The only disadvantage of this simple method is that you have to free your allocated value from the C side of your program. Usually this means providing a function to free the allocated memory with interface to that on the OCaml side. In case of a library this is equivalent to using the destroy function associated with a particular type.

```
typedef struct _obj_st {
    double d;
    int i;
    char c;
} obj_st;

typedef obj_st *obj_p;


CAMLprim value
wrapping_ptr_ml2c( value d, value i, value c )
{
    obj_p my_obj;
    my_obj = malloc(sizeof(obj_st));
    my_obj->d = Double_val(d);
    my_obj->i = Int_val(i);
    my_obj->c = Int_val(c);
    return (value) my_obj;
}

CAMLprim value
dump_ptr( value ml_ptr )
{
    obj_p my_obj;
    my_obj = (obj_p) ml_ptr;
    printf(" d: %g\n i: %d\n c: %c\n",
            my_obj->d,
            my_obj->i,
            my_obj->c );
    return Val_unit;
}

CAMLprim value
free_ptr( value ml_ptr )
{
    obj_p my_obj;
    my_obj = (obj_p) ml_ptr;
    free(my_obj);
    return Val_unit;
}
```

Then the pointer can be hidden in an abstract type on the OCaml side:

```
type t
external abs_get: float -> int -> char -> t = "wrapping_ptr_ml2c"
external print_t: t -> unit = "dump_ptr"
external free_t: t -> unit = "free_ptr"
```

```
# let ty = abs_get 255.9 107 'K' in
  print_t ty;
  free_t ty;
  ;;
 d: 255.9
 i: 107
 c: K
- : unit = ()
```

Finalisation of C objects

To finalise those abstract type that represent C object, don't try to use the `Gc.finalise` like this:

```
let abs_get f i c =
  let t = abs_get f i c in
  Gc.finalise free_t t;     (* doesn't work *)
  (t)
;;
```

because this function does only work with heap-allocated values.
But you can bypass this by creating a new type that mixes the abstract type with heap-allocated value:

```
type u = {t:t; s:string}

let free_t v = free_t v.t ;;
let print_t v = print_t v.t ;;

let abs_get f i c =
  let t = abs_get f i c in
  let u = {t=t; s=" "} in
```

```
    Gc.finalise free_t u;
    (u)
  ;;
```

This is a simple way, but there is an other "more official" way to achieve the same effect from the C side, which is described in the next paragraph.

Custom Operations Structure

You can use the function `caml_alloc_custom()` to store datas of a given size. The size of this data is given as the second parameter of this function. The first parameter is a `custom_operations` structure, which can be used to provide functions associated with this ocaml value. In the example below no functions are associated, so the `custom_operations` structure is filled with the default functions (which are in fact defined as `NULL` in `<caml/custom.h>`)

```
#include <caml/custom.h>
#include <string.h>

typedef struct _obj_st {
    double d;
    int i;
    char c;
} obj_st;

static struct custom_operations objst_custom_ops = {
    identifier: "obj_st handling",
    finalize:    custom_finalize_default,
    compare:     custom_compare_default,
    hash:        custom_hash_default,
    serialize:   custom_serialize_default,
    deserialize: custom_deserialize_default
};

static value copy_obj( obj_st *some_obj )
{
    CAMLparam0();
    CAMLlocal1(v);
    v = caml_alloc_custom( &objst_custom_ops, sizeof(obj_st), 0, 1);
    memcpy( Data_custom_val(v), some_obj, sizeof(obj_st) );
    CAMLreturn(v);
}

CAMLprim value
get_finalized_obj( value d, value i, value c )
{
    CAMLparam3( d, i, c );
    CAMLlocal1(ml_obj);

    obj_st my_obj;
    my_obj.d = Double_val(d);
    my_obj.i = Int_val(i);
    my_obj.c = Int_val(c);

    ml_obj = copy_obj( &my_obj );

    CAMLreturn(ml_obj);
}

CAMLprim value
access_obj( value v )
{
    obj_st * my_obj;
    my_obj = (obj_st *) Data_custom_val(v);
    printf(" d: %g\n i: %d\n c: %c\n",
            my_obj->d,
            my_obj->i,
            my_obj->c );
    return Val_unit;
}

type obj
external new_obj: float -> int -> char -> obj = "get_finalized_obj"
external dump_obj: obj -> unit = "access_obj"
```

The values optained by the function `new_abs` won't need an explicite free as in the previous paragraph Pointers to C structures, OCaml will finalise these values when the garbage collection will occur.

```
# let a = Array.init 20 (fun i -> new_obj (float i) i 'A') in
  dump_obj a.(9);
  ;;
 d: 9
 i: 9
 c: A
- : unit = ()
```

Custom Finalisation

Very often C structures contain allocated members, in such case you have to use a custom finalisation function to free these members.
You can see this in the exemple below which is very close to the previous example. Here the member `str` of the C structure is allocated, and needs to be freed in the custom finalisation function:

```
typedef struct _fobj {
    double d;
    int i;
    char * str;
} fobj;

void finalize_fobj( value v )
{
    fobj * my_obj;
    my_obj = (fobj *) Data_custom_val(v);
    free( my_obj->str );
    puts("fobj freed done");
}

static struct custom_operations fobj_custom_ops = {
    identifier: "fobj handling",
    finalize:  finalize_fobj,
    compare:     custom_compare_default,
    hash:        custom_hash_default,
    serialize:   custom_serialize_default,
    deserialize: custom_deserialize_default
};

static value copy_fobj( fobj *some_obj )
{
    CAMLparam0();
    CAMLlocal1(v);
    v = caml_alloc_custom( &fobj_custom_ops, sizeof(fobj), 0, 1);
    memcpy( Data_custom_val(v), some_obj, sizeof(fobj) );
    CAMLreturn(v);
}

CAMLprim value
get_finalized_fobj( value d, value i, value str )
{
    CAMLparam3( d, i, str );
    CAMLlocal1(ml_obj);
    int len;
    fobj my_obj;

    my_obj.d = Double_val(d);
    my_obj.i = Int_val(i);

    len = caml_string_length(str) + 1;
    my_obj.str = malloc( len * sizeof(char) );
    memcpy( my_obj.str, String_val(str), len );

    ml_obj = copy_fobj( &my_obj );

    CAMLreturn(ml_obj);
}

CAMLprim value
access_fobj( value v )
{
    fobj * my_obj;
    my_obj = (fobj *) Data_custom_val(v);
    printf(" d: %g\n i: %d\n str: %s\n",
            my_obj->d,
            my_obj->i,
            my_obj->str );
    return Val_unit;
}

type fobj
external new_fobj: float -> int -> string -> fobj = "get_finalized_fobj"
external dump_fobj: fobj -> unit = "access_fobj"
```

You can see when values are finalised with the `printf()` message. In the script below calling `Gc.full_major` will produce the finalisation:

```
let () =
  begin
    let f i = new_fobj (float i) i (String.make i '.') in
    let a = Array.init 20 f in
    dump_fobj a.(9);
  end;
  Gc.full_major ();
  print_endline "end test";
;;
```

You can find more informations about the custom_operations structure on the page http://caml.inria.fr/pub/docs/manual-ocaml/manual033.html#s:custom in section **19.9.1** in the offical manual.
This structure can be used too for other custom operations like *serialisation* and so on.

Variants with arguments

Now the case of variants with parameters, as in this example:

```
type pvar =
  | P0_a
  | P1_a of int
  | P2_a of int * int
  | P0_b
  | P1_b of int
  | P2_b of int * int

external handle_pvar: pvar -> unit = "param_variant"
```

OCaml values seen from on the C side, can be divised in two cathegories, longs and blocks. Booleans, characters, integers, and in the present case scalar variants are represented as a C long. Other types are blocks. It is possible to test from which cathegory belongs a value with these macros:

```
  Is_long(v)
  Is_block(v)
```

So you can use these to test if a variant is a scalar, or if it has parameters.

```
CAMLprim value
param_variant( value v )
{
    if (Is_long(v))
    {
        switch (Int_val(v))
        {
            case 0: printf("P0_a\n"); break;
            case 1: printf("P0_b\n"); break;
            default: caml_failwith("variant handling bug");
        }
    }
    else // (Is_block(v))
    {
        switch (Tag_val(v))
        {
        case 0: printf("P1_a(%d)\n",     Int_val(Field(v,0)) ); break;
        case 1: printf("P2_a(%d, %d)\n", Int_val(Field(v,0)), Int_val(Field(v,1)) ); break;
        case 2: printf("P1_b(%d)\n",     Int_val(Field(v,0)) ); break;
        case 3: printf("P2_b(%d, %d)\n", Int_val(Field(v,0)), Int_val(Field(v,1)) ); break;
        default: caml_failwith("variant handling bug");
        }
    }
    return Val_unit;
}
```

Here you can notice that the constructor of variants with arguments can be handled with the macro `Tag_val(v)`. Also be careful with the numbers to switch on, scalars and blocks are numbered separately as you can see in this example.

```
#load "funs.cma" ;;
open Funs ;;

let () =
  handle_pvar (P0_a);
  handle_pvar (P0_b);
  handle_pvar (P1_a(21));
  handle_pvar (P1_b(27));
  handle_pvar (P2_a(30, 34));
  handle_pvar (P2_b(70, 74));
;;
```

The type 'a option

```
type 'a option = None | Some of 'a
```

The type `'a option` is nothing more than a variant with one argument (as seen in the previous section), but here is the code as quick reference:

```
#define Val_none Val_int(0)

static value
Val_some( value v )
{
    CAMLparam1( v );
    CAMLlocal1( some );
    some = caml_alloc(1, 0);
    Store_field( some, 0, v );
    CAMLreturn( some );
}

CAMLprim value
rand_int_option( value unit )
{
    int d = random() % 4;
    if (d)
      return Val_some( Val_int(d) );
    else
```

```
      return Val_none;
}

external rand_int: unit -> int option = "rand_int_option"

# Array.init 10 (fun _ -> rand_int()) ;;
[|Some 3; None; Some 1; None; Some 3; Some 3; Some 2; None; None; Some 3|]
```

and in the opposite way:

```
external say: string option -> unit = "maybe_say"

#define Some_val(v) Field(v,0)

CAMLprim value
maybe_say( value speech )
{
    if (speech == Val_none)
        printf("Nothing\n");
    else
        printf("Something: %s\n", String_val(Some_val(speech)) );
    return Val_unit;
}

# say None ;;
Nothing
- : unit = ()

# say (Some "Camelus bactrianus") ;;
Something: Camelus bactrianus
- : unit = ()
```

Optional labeled parameter

In a very close way, you can put optional parameters using labels:

```
external say_lbl: ?speech:string -> unit -> unit = "maybe_say_label"
```

Note that here the type is `string` and not `string option`, and that from the C side it is seen as a `string option`.

```
CAMLprim value
maybe_say_label( value speech, value unit )
{
    if (speech == Val_none)
        printf("Nothing\n");
    else
        printf("Something: %s\n", String_val(Some_val(speech)) );
    return Val_unit;
}
```

You will then often need to add a unit parameter at the end of the parameters list, which also need to be added in the C function even if it is not used.

```
# say_lbl () ;;
Nothing
- : unit = ()

# say_lbl ~speech:"le chameau songeur" () ;;
Something: le chameau songeur
- : unit = ()
```

Polymorphic Variants

```
type plm_var = [
  | `plm_A
  | `plm_B
  | `plm_C
  ]

external plm_variant: plm_var -> unit = "polymorphic_variant"
```

The C representation of a polymorphic variant can by retrieved with a function that computes its hash value:

```
CAMLprim value
polymorphic_variant( value v )
{
    if (v == caml_hash_variant("plm_A"))  puts("polymorphic variant A");
    if (v == caml_hash_variant("plm_B"))  puts("polymorphic variant B");
    if (v == caml_hash_variant("plm_C"))  puts("polymorphic variant C");
    return Val_unit;
}

let () =
  plm_variant `plm_A;
  plm_variant `plm_B;
  plm_variant `plm_C;
;;
```

If performance is an issue, instead of make a call to `caml_hash_variant()` each time, what you can do is to get its result for each variant, and create constants which you use in a switch:

```
#include <caml/mlvalues.h>
#include <stdio.h>

int main()
{
    printf("#define  MLVAR_plm_A  (%d)\n", caml_hash_variant("plm_A") );
    printf("#define  MLVAR_plm_B  (%d)\n", caml_hash_variant("plm_B") );
    printf("#define  MLVAR_plm_C  (%d)\n", caml_hash_variant("plm_C") );
    return 0;
}
```

Which you can compile with:

```
> empty.ml
ocamlc -o empty.o -output-obj empty.ml
ocamlc -c variant.c
gcc -o variant.exe variant.o empty.o -L"`ocamlc -where`" -lcamlrun -lm -lcurses
```

`./variant.exe` will output this result:

```
#define  MLVAR_plm_A   (-1993467801)
#define  MLVAR_plm_B   (-1993467799)
#define  MLVAR_plm_C   (-1993467797)
```

Which you can then include at the beginning of your C code, then you can replace the previous function `polymorphic_variant()` by:

```
CAMLprim value
polymorphic_variant( value v )
{
    switch (v)
    {
        case MLVAR_plm_A:  puts("polymorphic variant A");  break;
        case MLVAR_plm_C:  puts("polymorphic variant B");  break;
        case MLVAR_plm_D:  puts("polymorphic variant C");  break;
        default:
            caml_failwith("unrecognised polymorphic variant");
    }
    return Val_unit;
}
```

And another way to get the same defines:

```
CAMLprim value
print_polymorphic_variant_val( value v )
{
    printf("%d", (long) v );
    fflush(stdout);
    return Val_unit;
}

external pmvar_print_i: pmvar -> unit = "print_polymorphic_variant_val"

let () =
  let p = Printf.printf in
  p "#define  MLVAR_plm_A \t %!";  (pmvar_print_i `plm_A);  p "\n%!";
  p "#define  MLVAR_plm_B \t %!";  (pmvar_print_i `plm_B);  p "\n%!";
  p "#define  MLVAR_plm_C \t %!";  (pmvar_print_i `plm_C);  p "\n%!";
;;
```

Notice the flush of the stdout channels on the two sides after each strings (in OCaml flush is achieved by `"%!"`), this is because the stdout of OCaml and C are two different channels that are not synchronised.

Linking against a library

Now imagine that you want to use a C library in the C part, let's say a fictitious library called "MyLib", which is compiled in C programs this way:
`cc -o my_prog -L/lib/path -lMyLib my_prog.c`
In this case the argument `-lMyLib` will have to be inserted while compiling the module, as shown below:

```
dllwrap_stubs.so: wrap.o
        ocamlmklib  -o  wrap_stubs  $<  \
            -L/lib/path  -lMyLib

funs.cmxa:  funs.cmx  dllwrap_stubs.so
        ocamlopt -a  -o $@  $<  -cclib -lwrap_stubs \
            -ccopt -L/lib/path  \
            -cclib -lMyLib

funs.cma:  funs.cmo  dllwrap_stubs.so
        ocamlc -a  -o $@  $<  -dllib -lwrap_stubs \
            -ccopt -L/lib/path  \
            -cclib -lMyLib
```

For the native code module, the directives have to be preceded by `-cclib` for the linker, and `-ccopt` for the compiler and linker. Note the use of `-dllib` for ocamlc.

Read the related manual pages for more informations:
the page about **ocamlc** http://caml.inria.fr/pub/docs/manual-ocaml/manual022.html in **Chapter 8**, and
the page about **ocamlopt** http://caml.inria.fr/pub/docs/manual-ocaml/manual025.html in **Chapter 11**.

Call Caml functions from C

To call caml functions from C, you have to give a string that identifies each caml functions with `Callback.register`. You can then retrieve this caml function from C with `caml_named_value("ID")`. And you can cache the call to `caml_named_value()`, as you can see in the C code below, with a `static` variable. But you still have to test if it is equal to `NULL` in case the caml garbage collector has freed it.

```
let print_hello () =
  print_endline "Hello World";
;;

let () =
  Callback.register "Hello callback" print_hello;
;;
```

```
#include <caml/callback.h>

void hello_closure()
{
    static value * closure_f = NULL;
    if (closure_f == NULL) {
        closure_f = caml_named_value("Hello callback");
    }
    caml_callback(*closure_f, Val_unit);
}
```

Start-up from C

If you want to make your main program from C, which will call caml parts, you can do it as below,
file `"ml_part.ml"`:

```
let print_hello () =
  print_endline "Hello World";
;;

let () =
  Callback.register "Hello callback" print_hello;
;;
```

file `"main.c"`:

```
#include <caml/mlvalues.h>
#include <caml/callback.h>

void hello_closure()
{
    static value * closure_f = NULL;
    if (closure_f == NULL) {
        closure_f = caml_named_value("Hello callback");
    }
    caml_callback(*closure_f, Val_unit);
}

int main(int argc, char **argv)
{
    caml_main(argv);
    hello_closure();
    return 0;
}
```

In the `main` function the first instruction have to be `caml_main(argv)` in order to init the caml part of the program. All the caml instructions at the root level (for exemple `print_endline "something" ;;`, and everything defined under `let () = (* code *) ;;`) will be evaluated and executed at this moment.
Compilation:

```
ocamlopt -output-obj ml_part.ml -o ml_part_obj.o

gcc -c main.c  -I"`ocamlc -where`"

gcc -o prog.opt  \
        main.o  ml_part_obj.o   \
        -L"`ocamlc -where`"     \
        -lm -ldl -lasmrun
```

The last line mentions the `-lm` and `-ldl` libraries, as can be found by:
`grep NATIVECCLIBS `ocamlc -where`/Makefile.config`
So for portability issue with your Makefile, it is possible to replace the two arguments `-lm -ldl` by the variable `$(NATIVECCLIBS)`. To achieve this you just need to include the file `` `ocamlc -where`/Makefile.config `` as below:

```
prog.opt: main.o  ml_part_obj.o
        gcc -o $@  $^   \
            -L"`ocamlc -where`"     \
            $(NATIVECCLIBS) -lasmrun

 -include  $(shell ocamlc -where)/Makefile.config
```

The related sections in the OCaml manual are here:
sections **19.7.4 Main program in C**, and **19.7.5 Embedding the OCaml code in the C code** at *Chapter 19* page
http://caml.inria.fr/pub/docs/manual-ocaml/manual033.html

Mixing with C++

Contents of file **mymod_stubs.cc**:

```
#include <iostream>
#include <string>

extern "C" {
#include <caml/memory.h>
#include <caml/mlvalues.h>
}

extern "C" value my_hello_cc (value v_str) {
  CAMLparam1 (v_str);

  std::cout << "Hello " << String_val(v_str) << "!\n";

  CAMLreturn (Val_unit);
}
```

Contents of file **mymod.ml**:

```
external my_hello: string -> unit = "my_hello_cc"
```

Contents of file **caller.ml**:

```
let _ =
  Mymod.my_hello "Blue Camel";
;;
```

compile to native code with:

```
g++ -o mymod_stubs.o -I`ocamlc -where` -c mymod_stubs.cc
ocamlopt -c mymod.ml
ocamlmklib -o mymod mymod_stubs.o
ocamlmklib -o mymod mymod.cmx

ocamlopt -I . -cclib -lstdc++ mymod.cmxa caller.ml -o caller.opt
```

then call it:

```
% ./caller.opt
Hello Blue Camel!
```

now the compilation for bytecode:

```
g++ -o mymod_stubs.o -I`ocamlc -where` -c mymod_stubs.cc
ocamlc -c mymod.ml
ocamlmklib -o mymod -lstdc++ mymod_stubs.o
ocamlmklib -o mymod  mymod.cmo

ocamlc -I . mymod.cma  caller.ml  -o caller.byte
```

then call it:

```
% ocaml mymod.cma caller.ml
Hello Blue Camel!

% ocamlrun -I . caller.byte
Hello Blue Camel!
```

This section about C++ was greatly inspired by the tutorial by *Anne Pacalet* (in French) which you can find on one of these web page:
http://anne-pacalet.developpez.com/tutoriels/ocaml/interface-c-ou-cpp-et-ocaml/
http://www-sop.inria.fr/everest/personnel/Anne.Pacalet/camltop.php

This was an introduction tutorial, for more complete details, refer to the official OCaml manual at Chapter 19 *Interfacing C with OCaml*, page:
http://caml.inria.fr/pub/docs/manual-ocaml/manual033.html
If there are questions that have not found an answer on this page nor in the official manual, you can join the **ocaml_beginners**

mailing list to find help, at this address:
http://tech.groups.yahoo.com/group/ocaml_beginners/
You may also find examples to follow from the **other ocaml bindings** registered in the OCaml-Hump:
http://caml.inria.fr/cgi-bin/hump.en.cgi?browse=65
If you are able to read French, I would also recommand you to read **this very good tutorial**:
http://www.siteduzero.com/tutoriel-3-187906-introduction-au-dialogue-entre-ocaml-et-le-c.html
Still in the French language, there is also these 2 pages on the Epita's wiki:
http://wiki-prog.infoprepa.epita.fr/index.php/20110307:TP:C:OCaml
http://wiki-prog.infoprepa.epita.fr/index.php/Programmation:C:OCaml
And I don't know if it's worth (I never used it), but just be aware that SWIG provides **support for OCaml**:
http://www.swig.org/Doc2.0/Ocaml.html

If you have saved this document, you can check for up-dates or corrections from **its original location on the web**:
http://www.linux-nantes.org/%7Efmonnier/OCaml/ocaml-wrapping-c.php

---

Version of this document

2013-03-18

Thanks to Eric Norige, for corrections of the English.
This document has been modified after Eric Norige English corrections, please write to me for other corrections.

You can send any kind of comments to me. My email is `monnier.florent on gmail`