Computer Labs: OO-programming with C
Or C vs. C++

2º MIEIC

Pedro F. Souto (pfs@fe.up.pt)

November 23, 2018

Object Oriented Programming

- Object-oriented programming is a programming paradigm that facilitates the development of programs in general and of some classes of programs in particular:
  - Graphical user interfaces (GUI)
  - Computer graphics programs (e.g. games)
  - Computer simulations
- C, unlike C++, is not object-oriented
  - However, it is possible to develop code in object-oriented style with C
  - Actually, to be more precise we should say:
    - It is possible to develop **abstract data types** in C

OO and Classes (A Short and Simplistic View)

- A language is OO if it supports the concept of **Class**
- A class is a (data-)type that includes both

  State i.e. data members/fields

  Behavior i.e. functions/methods that operate on the class's state
- An **object** is an instance of a class
- A key concept in OO is **data encapsulation** or **data hiding**, i.e.:
  - Access to the state of an object is possible only by invoking its methods

  Most of the advantages of object orientation stem from this property.

Problem How can we implement classes/objects in C in such a way that ensures data-encapsulation?

Specifying a Class in C

Idea Use:

`structs` to store the state of an object

functions as the methods of a class

- The first argument of each function should be a pointer to the state of the object on which the method will operate

- Define each class in its own C source (`.c`) file
  - This further contributes to the modularity and to the pluggability of the code
  - Helps ensuring data hiding (see below)

Example: A Queue Class – `queue.h`

```
typedef struct {
    int *buf;            // pointer to array that stores queue elements
    int in,out;          // indices of the array pointed by buf
                         //  to insert/remove elements
    int size;           // size of the array
    int count;           // number of elements in queue
} queue_t;

queue_t *new_queue(unsigned int ini_size); // "constructor":
                                     //    ini_size, initial queue size
void delete_queue(queue_t *q);       // queue destructor
int put_queue(queue_t *q, int n);    // enqueue 'n' in queue, returns
                                     //    0 in case of success
                                     //    -1 otherwise (queue full,
                                     //                    or no space)
int get_queue(queue_t *q, int *p);  // dequeue first element from queue
                                     //    0 in case of success
                                     //    -1 otherwise (queue empty)
```

- Note that for all "methods" but the constructor, the first argument specifies the object that is the target, i.e. the object on which the method will operate.

Parenthesis: `typedef`

- Defines a new data type name
  - E.g.

    ```
    typedef unsigned char uchar;
    ```

    makes `uchar` a synonym of `unsigned char`

  It does not define a new type
- POSIX recommends(?reserves?) that data type names have the `_t` suffix to distinguish them from other names
- There are two main reasons to use `typedef`:

  Portability E.g. the size of integer types may differ among architectures/compilers

  Readability The name `queue_t` is clearer than using some complicated struct, although we might use `struct queue` instead

  besides aesthetics.

Example: Constructor and Destructor

C has no `new` nor `delete` instructions

Use `malloc()` and `free()` instead

- These are functions of the C standard library
- In addition to `malloc()` there is also `realloc()` (and `calloc()`)

```
#include <stdlib.h>
    ...
    int *buf, *ptr;
    ...
    // allocate an array for 100 ints
    buf = (int *) malloc(100*sizeof(int));
    ...
    // reallocate more space for 100 ints
    //   -- preserve the value of the first 100 ints
    ptr = (int *) realloc(buf, 100*sizeof(int));
    if( ptr == NULL ) { // run out of memory
        ...
    } else
        buf = ptr;
    ...
    free(buf);
```

Parenthesis: The `sizeof` Operator

- `sizeof` allows to compute the size in bytes of:
  - a variable, an array or structure;
  - a type i.e. a basic type, or a derived type such as a structure or pointer

```
typedef struct {int color, char *msg} Msg;
int size = sizeof(Msg); // size of struct in bytes
Msg msgs[20];
int size_msgs = sizeof(msgs); // size of array in bytes
// number of elements in array
// int num_el = msgs.size(); // in C++
int num_el = sizeof(msgs)/sizeof(Msg) ;
// dynamically allocate an arry with N elements
// Msg *mptr = new Msg[N]; // in C++
Msg *mptr = (Msg *) malloc(N*sizeof(Msg));
// free memory
// delete[] mptr; // in C++
free(mptr)
```

Example: Queue Implementation – `queue.c`

```
queue_t *new_queue(unsigned int in_size) {
    // allocate queue object
    queue_t *q = malloc(sizeof(*q));
    if( q == NULL )
        return NULL;
    // allocate space to store queue elements
    q->size = in_size ? in_size : 1;
    q->buf = malloc(q->size * sizeof(int));
    if( q->buf == NULL ) {
        free(q);
        return NULL;
    }
    // initialize state of queue
    q->in = q->out = q->count = 0;

    return q;
}
void delete_queue(queue_t *q) {
    free(q->buf);
    free(q);
}
```

Example Queue Implementation – queue.c

```
int put_queue(queue_t *q, int n) {
    if( q->count == q->size )
        if( resize_queue(q) ) // private function
            return -1;
    q->buf[q->in++] = n;
    q->count++;
    adjust_queue(q); // private function
    return 0;
}
int get_queue(queue_t *q, int *n) {
    if( q->count != 0 ) {
        *n = q->buf[q->out++];
        q->count--;
        adjust_queue(q);
        return 0;
    }
    return -1;
}
```

Question: How to ensure that `resize_queue()` and `adjust_queue()` are private?

- In C, by default, all functions are global, i.e. public

Answer: Use the `static` Keyword

- The static keyword limits the scope of an “object”, i.e.
  function
  variable
  to the C source (`.c`) file where that “object” is defined
  - The `static` keyword provides a means of hiding **names** of global objects from other modules, i.e. C source files

```
// private: can be invoked only in queue.c
static void adjust_queue(queue_t *q) {
    q->in %= q->size;
    q->out %= q->size;
}
static int resize_queue(queue_t *q) {
    int *p = (int *)realloc(q->buf, 2*(q->size)*sizeof(int));
    int i;
    if( p == NULL )
        return -1;
    q->buf = p;
    for( i = 0; i < q->in; i++)
        q->buf[q->size + i] = q->buf[i];
    q->in += q->size;
    q->size *= 2;
    return 0;
}
```

Parenthesis: More on `static`

- When applied to **local variables**, i.e. variables defined inside a function, `static` means that that variable and its value persist between invocations of that function

Static Global Variables vs. Global Variables

foo.c:

```
int totallyGlobal;
static int locallyGlobal;
void foo() {
     totallyGlobal = 1;
     locallyGlobal = 2;
}
```

bar.c:

```
extern int totallyGlobal;

void bar() {
     totallyGlobal = 1;
}
```

Private Functions

popo.c:

```
// invokable only in popo.c
static void popo() {
    ...
}
```

Persistent Local Variables

xpto.c:

```
// counts number of xpto() invocations
void xpto() {
     static int count = 0;
     count++;
     ...
}
```

Ensuring Encapsulation

- **Encapsulation** hides the details of implementation of an object from its users
- The use of private methods by means of `static` is not enough:
  - In our implementation, a user can access any field of the object using a pointer to the corresponding struct:

    ```
    q->size += 10;
    ```

Question: How can we prevent it?

Answer: Hiding the implementation of the `queue_t`

`queue.h`

```
struct queue;
typedef struct queue queue_t
```

The “class”’ user needs pointers to `queue_t`, thus there is no problem if the type is **incomplete**

`queue.c`

```
#include "queue.h"
struct queue {
    char *buf;
    int in, out;
    int size, count;
};
```

Only the “class” implementation needs to know the data members of `struct queue`

Example: Use of Queue

```
#include "queue.h"

int main(int argc, char *argv[]) {
    queue_t *q;
    char *end_ptr;
    unsigned int size = 20;   // queue default size
    int n;

    if( argc == 2 )
        size = strtoul(argv[1], &endptr, 10);

    if( endptr == argv[1] )
     return -1;

    if( (q = new_queue(size)) == NULL )
         return -1;
    if( put_queue(q, 77) != 0 )
        printf("Queue full\n");
    if( get_queue(q, &n) != 0 )
        printf("Queue empty\n");
    else
        printf("Dequeued %d\n", n);
    delete_queue(q);
    return 0;
}
```

Generic “Classes”

Problem: `queue_t` (or `struct queue`) is able to store values of type `int` only

- To store values of other types we could write a different class

Question How about to implement something like C++ templates?

Answer Yes, we can

- All we need is to use generic pointers, i.e. `void *`
- But ... we cannot take advantage of pointer arithmetic

Parenthesis: Pointer Arithmetic

- A C pointer is a data type whose values are memory addresses of variables of a given type
- In C, the name of an array is the address of the first element of that array:

```
int a[5];
p = a;          /* set p to point to the first element *
p = & (a[0]); /* same as above */
```

- Conversely, we can use the “array notation” to refer to element `i` of array `a`;

```
for( i = 0; i < 5; i++) {
      p[i] = 0;
}
```

- C supports pointer arithmetic – meaningful only when used with arrays:

```
for( i = 0; i < 5; i++, p++) {
      *p = 0;
}
```

- In the implementation of `queue_t`, we used the array notation to access the elements in the queue. E.g. in `put_queue()`:

```
q->buf[q->in++] = c;
```

Example: Generic Queue

- Because we are using generic pointers we cannot rely on the C compiler for pointer arithmetic:
  - The compiler does not know the size of each element in the queue
  - The size of each element must be kept as part of the state of the generic queue

```
#include "gqueue.h"
struct gqueue {
    void *buf;          // void * instead of int *
    int in, out;
    int size, count;
    int el_size;        // for pointer arithmetic
};
```

Question: What is the meaning of `in` and `out` (`size` and `count`)?

Example: Generic Queue

Alternative I: Same meaning as in `queue_t`

in index of element in array pointed to by `buf`

out index of element in array pointed to by `buf`

This is the alternative closer to what the C compiler does when a pointer to a type is used

Alternative II

in offset of element in array pointed to by `buf`

out offset of element in array pointed to by `buf`

In this casem it might be better to name the members `in_off` and `out_off`

Alternative III

in pointer to position in array pointed to by `buf`

out pointer to position in array pointed to by `buf`

It would have been better to define `in` and `out` as `void *`

Example: Generic Queue – gqueue.c

```
gqueue_t * new_gqueue(unsigned int n_el, int el_size) {
    gqueue_t * q = malloc(sizeof(gqueue_t));
    if( q == NULL )
        return q;
    // The user must provide the size of each queue element
    q->size = n_el ? n_el : 1;
    q->buf = malloc(q->size * el_size);
    if( q->buf == NULL ) {
        free(q);
        return NULL;
    }
    q->in = q->out = q->count = 0;
    q->el_size = el_size;
    return q;
}
void delete_gqueue(gqueue_t *q) {
    free(q->buf);
    free(q);
}
int is_empty_gqueue(gqueue_t *q) {
    return q->count == 0;
}
```

Example: Generic Queue – gqueue.c

```
int is_full_gqueue(gqueue_t *q) {
    return q->count == q->size;
}
int put_gqueue(gqueue_t *q, void *el) {
    if( is_full_queue(q) )
        return -1;
    // memcpy(dst, src, n_bytes): memory copy
    // must do pointer arithmetic explicitly
    memcpy(q->buf + q->in*q->el_size, el, q->el_size);
    q->in = (q->in + 1) % q->size;
    q->count++;
    return 0;
}
int get_gqueue(gqueue_t *q, void *el) {
    if( is_empty_gqueue(q) )
        return -1;
    memcpy(el, q->buf + q->out*q->el_size, q->el_size);
    q->out = (q->out + 1) % q->size;
    q->count--;
    return 0;
}
```

Example: Use of Generic Queue

```
typedef struct {int time, freq;} note_t;

gqueue_t *nq = new_gqueue(10, sizeof(note_t));

note_t in, on;
for( i = 0; i<30; i++) {
    in.time = 1; in.freq = (i+2)*10;
    if( put_gqueue(nq, &in) != 0 )
        printf("Full queue\n");
    if( get_gqueue(nq, &out) == 0 ) {
        printf("%d-%d \n", on.time, on.freq;
    } else {
        // This should never occur
        printf("Empty queue\n");
    }
}
delete_gqueue(nq);
```

Conclusion

- It is possible to use C, thinking in C++
- However:
  - C is not C++
  - You need more discipline to structure your program and write your code
- We expect you to apply these concepts in your project
  - If you need some well known data structure (queue, stack, ...) take a look to the interfaces of the classes supported by OO languages, such as C++, Java or C#

Thanks to:

I.e. shamelessly translated material by:

- João Cardoso (jcard@fe.up.pt)