Rules for teaching C as a first programming language

John C. Lusth

Revision Date: April 12, 2017

Introduction

This document concerns the teaching of C as a first language in a CS1 course. As it is often the desire to include the ideas of object encapsulation in a first course, C should be taught in a style that emphasizes encapsulation. Students who learn this style should have little trouble in moving to an object-oriented language later, especially if the advanced ideas (last section) are taught.

The remainder of this document assumes a familiarity with C and some object oriented language.

The basic idea

Instructors should always stress the object oriented nature of the program, emphasizing that each class gets its own module. In general, the public interface of a class *X* is placed in a file named *X.h*, often called a *header file*. The public interface consists of the structure that encapsulates the components of the class and the prototypes of the public functions (methods) that (typically) operate upon the structure. The public interface also includes external declarations of any components that are shared between objects of a class. The implementation of the class *X* is placed in a file named *X.c*. The implementation consists of the function definitions and variable declarations that were mentioned in the public interface along with definitions and declarations that are private (i.e., not visible outside the class).

An example class declaration

Here is the public interface for a *node* class, which can be used to make a linked list. The public interface is stored in a file named *node.h*:

```
#ifndef __NODE_INCLUDED__
#define __NODE_INCLUDED__
typedef struct nodeobject node; /* forward declaration of our structure */

extern char *INTEGER;
extern char *REAL;
extern char *STRING;

extern node  *newNodeInteger(int value,node *next);
extern node  *newNodeReal(double value,node *next);
extern node  *newNodeString(char *value,node *next);
extern char  *getNodeType(node *n);
extern int    getNodeInteger(node *n);
extern double getNodeReal(node *n);
extern char  *getNodeString(node *n);
extern node  *getNodeNext(node *n);
#endif
```

We can have three kinds of nodes, one that stores an integer value, one that stores a real value, and one that stores a string value.

We also declare external the three character constants to populate the *type* field, so we know what kind of node we have. Note that it is not necessary to explicitly declare the methods of the class external, but we do so, so that the instance variables and the method prototypes look consistent in the *.h* file.

The public interface exposes three node creating functions, named *newXXX*, depending on what kind of node you would like to create. It also exposes the access functions, named *getXXX*, to retrieve the various instance variables of the node. You may also have mutators, usually named *setXXX*; this example has no need of mutators, however.

The implementation, which should be stored in a file named *node.c*, looks like this:

```
#include <stdio.h>
#include <stdlib.h>
#include "node.h"

struct nodeobject
    {
    char *type;
    int ival;
    double rval;    /* ival, rval, sval: one will hold the actual value */
    char *sval;
    node *next;
    };

static node *newNode(void);

/*************** public interface *************/

char *INTEGER = "integer";
char *REAL = "real";
char *STRING = "string";

/* constructors */

node *
newNodeInteger(int v,node *n)
    {
    node *p = newNode();
    p->type = INTEGER;
    p->ival = v;
    p->next = n;
    return p;
    }

node *
newNodeReal(double v,node *n)
    {
    node *p = newNode();
    p->type = REAL;
    p->rval = v;
    p->next = n;
    return p;
    }
```

```
node *
newNodeString(char *v,node *n)
    {
    node *p = newNode();
    p->type = STRING;
    p->sval = v;
    p->next = n;
    return p;
    }

/* accessors */

char  *getNodeType(node *n)    { return n->type; }
int    getNodeInteger(node *n) { return n->ival; }
double getNodeReal(node *n)    { return n->rval; }
char  *getNodeString(node *n)  { return n->sval; }
node  *getNodeNext(node *n)    { return n->next; }

/*************** private methods *************/

static node *
newNode()
    {
    node *n = (node *) malloc(sizeof(node));
    if (n == 0) { fprintf(stderr,"out of memory"); exit(-1); }
    return n;
    }
```

We can see from the implementation that our node structure has separate fields for each of the types our node can store. We could use a union to save some space, but these days of lots of memory, it's hardly worth the trouble.

We can now make and manipulate nodes:

```
#include <stdio.h>
#include "node.h"

int
main()
    {
    node *n;

    n = newNodeInteger(3,0); /* zero is the null pointer */
    n = newNodeReal(5.5,n);
    n = newNodeString("hello",n);

    while (n != 0)
        {
        char *t = getNodeType(n);
        if (t == INTEGER)
            printf("%d\n",getNodeInteger(n));
        else if (t == REAL)
            printf("%f\n",getNodeReal(n));
        else if (t == STRING)
            printf("%s\n",getNodeString(n));
```

```
        n = getNodeNext(n);
        }

    return 0;
    }
```

Here are links to this source code: nodetest.c, node.c, and node.h.

A more complicated example

The *X.h* file contains the typedef-ed structure representing class X as well as the prototypes for all the public methods. Consider a *Student* class which holds the ID number of a student, a list of the classes he or she is currently taking, a list of past classes, and the student's cumulative gpa. The file named *Student.h*, might look like:

```
#ifndef __STUDENT_INCLUDED__
#define __STUDENT_INCLUDED__
    #include <stdio.h>
    #include "class.h"

    typedef struct student_object Student;

    extern Student *newStudent(int id);
    extern void addStudentClass(Student *s, Class *c,int grade);
    extern void setStudentGrade(Student *s, Class *c,int grade);
    extern void setStudentGPA(Student *s, FILE *where);
    extern void displayStudent(Student *s, FILE *where);
#endif
```

Note the typedef that gives a simple name to a *Student* object. Note also that *Student* references another class named *Class*.

Public methods for a class follow a naming convention. For example, an insert method for objects of class *X* would be named *insertX*. Another convention is that the first argument for a class method is always the object.

Public versus private

Students should be well versed in the use of the keywords `static` and `extern` for controlling the visibility of definitions and declarations. Public components that are shared by all objects in a class are declared in the associated *.c* file. For example, suppose a shared component is the total number of students. This would be declared as a global in the *.c* file as:

```
int NumberOfStudents = 0;
```

if it is a public shared attribute. These public, shared components are also re-declared extern in the *.h* file, as in:

```
extern int NumberOfStudents;
```

If the component is to be shared, but made private, it would be declared static in the implementation file, as in:

```
static int NumberOfGoodStudents;
```

and would not appear in the header file.

Private methods in the implementation are placed in the *.c* file but are declared static.

Constructors

The constructor for a class X is named *newX*, by convention. It allocates space for the object and then initializes the components of the object. For example, if the student structure looked like:

```
struct student_object
    {
    int id;
    Class **currentClasses;
    int currentClasseCount;
    Class **pastClasses;
    int pastClassCount;
    double gpa;
    };
```

the constructor *newStudent* might look like:

```
Student *
newStudent(int id)
    {
    Student *p;

    /* allocate the object */

    p = (Student *) malloc (sizeof(Student));

    if (p == 0) Error("Allocating a Student: out of memory");

    /* initialize components */

    p->id = id;
    p->currentClasses = 0;
    p->currentClassCount = 0;
    p->pastClasses = 0;
    p->pastClassCount = 0;
    p->gpa = 0.0;

    ++NumberOfStudents;

    return p;
    }
```

Multiple constructors require unique names. By convention, constructors for class X would all have the form *newXalpha*, where *alpha* is either empty or a description that distinguishes the constructor from the other constructors.

Encapsulating methods

It is easy enough for an object to "carry along" methods as function pointers, relieving the programmer from coming up with unique names for public methods. This is probably too much for a first semester course, however. Also, realize that the object must still be passed to these methods so that the object's components are all accessible.

Here is the *Student* structure rewritten in such a style:

```
#include "student.h"

struct student_object
    {
    /* components */
    int id;
    Class **currentClasses;
    int currentClassCount;
    Class **pastClasses;
    int pastClassCount;
    double gpa;

    /* methods */
    void (*addClass)(Student *s,Class *c,int grade);
    void (*setGrad)(Student *s,Class *c,int grade);
    void (*setGPA)(Student *s);
    void (*display)Student *s);
    } Student;

extern Student *newStudent(int id);
```

The constructor for Student sets the function pointers in the methods section of the typedef:

```
Student *
newStudent(int id)
    {
    Student *p;

    /* allocate the object */

    p = (Student *) malloc (sizeof(Student));

    if (p == 0) Error("Allocating a Student: out of memory");

    /* initialize components */

    p->id = id;
    p->currentClasses = 0;
    p->currentClassCount = 0;
    p->pastClasses = 0;
    p->pastClassCount = 0;
    p->gpa = 0.0;

    /* initialize methods */

    p->addClass = addClass;
    p->addGrade = addGrade;
    p->setGPA = setGPA;
    p->display = display;

    ++NumberOfStudents;

    return p;
    }
```

The methods *addClass*, *addGrade*, *setGPA*, and *display* are defined with the static keyword (making them private) in *Student.h*.

Here is an example call to the *display* method:

```
Student *s;

s = newStudent(123456789);
/* add past classes */
...
s->setGPA(s);
s->display(s);
```

Even when encapsulating methods, one still needs to pass the object itself to the method.

Inheritance

As inheritance is not allowed, code reuse is accomplished via wrapper functions that dispatch to clients. Instead of a class X inheriting from class Y, an object of type X instead has a component which holds a pointer to an object of type Y. Suppose inserting into an X object is really an insertion into the Y object. If so, the solution is to define a function *insertX* that simply calls *insertY* on object component Y.

Advanced ideas

In the above scenario, all of the components of an object are private, but the object itself is exposed and can be manipulated outside of the public interface. If the object itself must be private, then the constructors are reformulated to return an index into a private array of objects (similar to the *open* constructor for file descriptor objects). The object is now an integer rather than a pointer to a structure. The *newStudent* constructor, in this case, might look like:

```
int
newStudent(int id)
    {
    Student *p;

    ++NumberOfStudents;

    p = (Student *) malloc (sizeof(Student));

    if (p == 0) Error("out of memory");

    p->id = id;
    p->currentClasses = 0;
    p->currentClassCount = 0;
    p->pastClasses = 0;
    p->pastClassCount = 0;
    p->gpa = 0.0;

    return add(p);
    }
```

The *add* function resizes the array holding the allocated *Student* objects to make room for the new object and returns the index corresponding to where the new object was placed:

```
static int
add(Student *p)
    {
    /* increase the number of students */

    ++NumberOfStudents;

    /* reallocate where the object is stored */

    Students = (Student **) realloc(Students * sizeof(Student *)
        * NumberOfStudents);

    if (Students == 0) Error("out of memory");

    /* store the student object and return its index */

    Students[NumberOfStudents - 1] = p;
    return NumberOfStudents - 1;
    }
```

The *Student* array is also declared static in the *Student.c* file:

```
static Student **Students = 0;
```

Of course, the *add* routine should use array size doubling if a large amount of objects are to be allocated.

The downside to privatizing the components is that type checking on the first argument to class methods (the object argument) is weakened since the first argument is now an integer.