PROGRAMACIÓN ORIENTADA A OBJETOS EN LENGUAJES NO ORIENTADOS A OBJETOS: C, UNA EXPERIENCIA

RESUMEN
Son indudables los beneficios cuando se realiza Programación Orientada a Objetos: reusabilidad de código, herencia, polimorfismo, etc.. Existen innumerables lenguajes que soportan dicho paradigma, como C++, EIFFEL, JAVA, etc., pero aún se siguen desarrollando de manera importante, aplicaciones en lenguajes que no soportan la mencionada metodología, como por ejemplo C. Este artículo pretende introducir al programador de aplicaciones tradicionales en lenguaje C, en las técnicas y metodologías para desarrollar dichas aplicaciones usando Programación Orientada a Objetos.

JORGE IVAN RIOS PATIÑO
Profesor Ing. Sistemas y Computación – Universidad Tecnológica de Pereira
Ing. Industrial – U. Tecnológica de Pereira
Mc.S Informática e Ing. del Conocimiento – U. Politécnica. de Madrid
Ph.D (c) Informática – Universidad Politécnica de Madrid
jirios@utp.edu.co

ABSTRACT
The benefits are doubtless when Programation OO is made: reusabilidad of code, inheritance, polymorphism, etc.. Innumerable languages that support this paradigm, like C++, EIFFEL, JAVA, etc. exist, but continued developing of important way, applications in languages that do not support mentioned metodologies, as for example C. This article tries to introduce the traditional application programmer in language C, in the techniches and metodologies to develop these applications using OO Programation.

1. INTRODUCCIÓN

A pesar de que hoy día existen innumerables lenguajes que soportan la Programación Orientada Objetos (POO, de aquí en adelante), como C++, EIFFEL, aun dista mucho de que estos se conviertan en lenguajes de desarrollo permanente, como lo hacen lenguajes que no soportan directamente la POO, tales como C, ADA, FORTRAN, etc, y en donde hay una gran cantidad de código desarrollado y se siguen desarrollando aplicaciones importantes.

Este artículo pretende mostrar como es posible mediante lenguajes que no soporten la POO (en este caso concreto el lenguaje C), hacer desarrollos basados en este paradigma, producto de la experiencia vivida por el autor, cuando participó en los proyectos SAIH –Sistema Automático de Información Hidrológica– y CYRA –Cálculo y Razonamiento Aproximado– en el Laboratorio de Sistemas Inteligentes de la Universidad Politécnica de Madrid y la Fundación Agustín de Bethencouth, en donde se emplearon técnicas de POO para el desarrollo de los sistemas propuestos.

2. PROGRAMACIÓN ORIENTADA A OBJETOS VS. PROGRAMACIÓN BASADA EN OBJETOS

Muchas veces, aún empleando lenguajes que soporten la POO, se comete el error de no programar en dicha metodología, porque no hay conceptos claros acerca de la misma.

Se debe recordar, que cualquier sistema desarrollado en metodología POO, debe reunir al menos las siguientes características:

- El sistema ha sido diseñado, identificando claramente las clases y su jerarquía pertinente.
- Uso del polimorfismo
- Definición clara en cada clase de sus métodos
- Encapsulamiento de la información
- Modularización
- Excepciones y Concurrencia
- Asociación entre Objetos

Esto se confunde muchas veces, por que el sistema se desarrolla solamente identificando objetos, sin los claros aspectos enumerados anteriormente; por lo tanto esto no basta para que un sistema sea declarado que ha sido desarrollado bajo POO. Lo contrario simplemente sería una programación basada en objetos, dado que la mencionada disciplina tiene unas características bastante definidas, que deben ser implementadas y seguidas minuciosamente para que los programas puedan ser catalogados así.

3. PROGRAMACIÓN ORIENTADA A OBJETOS UTILIZANDO LENGUAJE C

Aunque ya habíamos indicado que el lenguaje C no es un lenguaje estrictamente orientado hacia la POO, se pueden seguir ciertas reglas, para usarlo rigurosamente bajo esta metodología.

Fecha de Recepción: 31 Mayo de 2005
Fecha de Aceptación: 15 Noviembre de 2005

Según Rambaugh [1], para implementar un diseño que corresponda a la POO, debe cumplirse, entre otros, los siguientes requisitos:

- Traducir las clases en estructuras de datos
- Pasar argumentos a los métodos
- Reservar espacio para los Objetos
- Implementar la herencia en estructura de datos
- Implementar la resolución de métodos
- Implementar las asociaciones
- Resolver la concurrencia
- Encapsular los detalles internos de las clases

Para lo anterior, C ofrece alternativas para realizar dichas implementaciones: comprobación débil de tipos, punteros a funciones, utilización de macros, asignación y liberación de memoria e implementación de estructuras de datos.

3.1 Definición de Clases

El concepto de Clase no existe como tal en C. Para ello se puede usar el mecanismo tipo de datos de estructura (**struct**) para definir y encapsular toda la información referente a la misma, es decir, sus miembros datos y métodos.

Por ejemplo, si se fuera a definir un conjunto de jerarquías para describir toda una taxonomía de un conjunto de recursos hidráulicos como ríos, lagos, pantanos, etc. (Ver Fig. No. 1.) empleando el concepto de estructura, podemos, en primera instancia, crear la clase superior o abstracta Recurso Hidráulico (Ver Fig No. 2)

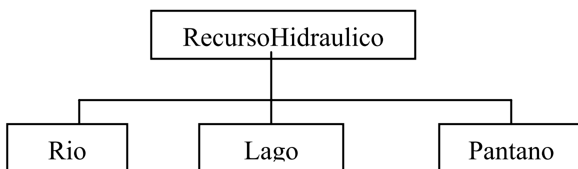

Figura No. 1

```
Typedef  Struct
     {      char      *Nombre;
            double    AreaCuenca;
            char      *CoordXnacimiento;
            char      *CoordYnacimiento;
            ..................

     } ClaseRecurso   *RecursoHidráulico ;
```

Figura 2 Ejemplo en C de definición de una Clase

La anterior definición permite, no solamente la declaración de una clase superior, si no que encapsula los atributos de la misma.

3.2 Herencia

La herencia en C, se implementa referenciando la estructura superior en la estructura que deriva. Es importante recalcar que cuando la herencia es múltiple, las referencias a las clases superiores se resuelven de la misma manera.

Por ejemplo (ver Fig. No. 2), si fuéramos a representar la subclase ***Rio***, deberemos declarar dentro de la misma estructura que lo define, un miembro (Rio) que haga referencia a la clase superior de la cual deriva: ***RecursoHidráulico***.

```
typedef  struct
     {  RecursoHidráulico       Recurso;
        Double                  Longitud;
          ..................
          …………
     }  ClaseRio          *Rio;
```

Figura 3 Ejemplo en C de la creación de una Subclase

La clase ***ClaseRio***, contendrá (*heredará*) todo lo referente a la clase superior ***RecursoHidráulico.***

El acceso a cada uno de los miembros de cada clase se puede hacer al estilo tradicional, como lo permite el lenguaje C, para este tipo de casos (Ver Fig. No. 4)

```
Rio → Longitud = 128.45;
```

Figura 4 Acceso a un Atributo

Aunque es totalmente válido, se recomienda que se definan métodos tanto de acceso como de escritura para cada uno de los datos miembros de cada clase. Esto se hace porque cuando las jerarquías se van volviendo complejas, el acceso también se da de la misma forma (ver Fig. No. 5).

```
RecursoHidraulico → Rio → Nombre = "Otun";
```

Figura 5 Acceso a un Atributo se una Subclase

Una forma de realizar dicha implementación es mediante el uso de macros. (ver Fig. No. 6).

```
#define RecursoHidráulico_Nombre(Recurso)
                ((RecursoHidráulico) Recurso → Nombre))

#define Rio_Longitud (R) ((Rio) R → Longitud))

#define Rio_E_Longitud (R, Long)
                         ((Rio) R → Longitud = Long))

#define Rio_Recurso (R) ((Rio) R → Recurso))

#define Rio_Nombre (R)
 ((Rio) RecursoHidráulico_Nombre(Rio_Recurso(R)))
```

Figura 6 Ejemplo de Acceso a Atributos por medio de Macros

Las ventajas de tener estos métodos, es que hay claridad en el código, lo hace funcional, fácil de depurar y sobre todo que se simula la protección de los datos miembro de las clases y sus objetos, dado que en C no es posible definir características explícitas de control (P*ublic*, *Private*, y *Protected*), sobre los mismos.

Como único mecanismo excepción a lo anterior, es que se pueden definir variables (datos miembro) de manera estática (***static***), lo cual simularía la característica explícita de control *Private*.

3.3 Resolución de Métodos

Los métodos inherentes a una clase se implementan mediante la declaración de los mismos dentro de la estructura (*clase*) al cual pertenecen, lo que permite que los mismos se resuelvan en el momento de la compilación

De igual forma que los datos de las clases, los métodos de cada clase no tienen mecanismos de control de acceso a los mismos, debido a que C carece explícitamente de ellos. Por lo tanto la resolución de los métodos se realiza utilizando punteros a funciones, incluyendo aquellos que simulan los que *construyen* (constructores) y *destruyen* (destructores) la clase: (Ver Fig. No. 7).

```
typedef struct
    {       RecursoHidráulico       Recurso;
            Double                  Longitud;
            .................
            void (* CosntructorRio) ();
            void (* DestructorRio)  ();

            double (* AreaCuenca) ();
            double (* Longitud)   ();
           …………
    } ClaseRio    *Rio;
```

Figura 7 Ejemplo de Resolución de Métodos en C

Se debe notar, que la clase *ClaseRio* hereda todos los métodos de su clase superior, en este caso *RecursoHidraulico*, dado que todo descriptor de clase contiene las operaciones de su superclase [2].

Mas aun, la clase derivada puede redefinir los métodos de su clase superior, inclusive apelando a los mismos nombres, sin que surja con esto algún conflicto, dado que la gestión de los mismos, está dada en espacios diferentes. La única precaución que hay que tener, es la forma de invocar los métodos, pero para ello se puede apelar a las macros, precisamente para eliminar cualquier confusión, sobre todo en la gestión del espacio de nombres.

3.4 Asociaciones

La implementación de las relaciones de asociación entre clases u objetos de las mismas, se realizan, a excepción de JAVA, de la misma forma que en la mayoría de los lenguajes orientados hacia la POO: usando punteros.

Por ejemplo, la asociación binaria se implementa como un dato miembro en cada una de los objetos asociados, dato que contiene un puntero al objeto asociado.

Por ejemplo, si se quisiera tener una asociación entre un *Rio* y sus *Afluentes*, *uno-a-muchos*, esta se podría implementar de la siguiente manera (ver Fig No. 8):

```
typedef struct ClaseRio
    {       RecursoHidráulico       Recurso;
            Double                  Longitud;
            .................
            void (* ConstructorRio) ();
            void (* DestructorRio)  ();
            double (* AreaCuenca) ();
            double (* Longitud)   ();
           …………
           ClaseRio        *Afluente;
    }      *Rio;
```

Figura 8 Ejemplo de Implementación de Asociaciones

En caso de que sean mas los objetos relacionados, las asociaciones se pueden implementar por medio de otras estructuras mas complejas como vectores, árboles, matrices tipo esparcida (*sparce*), etc.; y cuando las relaciones por efecto de la jerarquía de clases, se puedan volver complejas, es recomendable crear funciones que realicen de manera automática dicha operación, simulando por ejemplo, los mecanismos que se tiene en LISP para la manipulación de listas [2].

3.5 Creación y Destrucción de Objetos

La creación y destrucción de objetos, se resuelve en tiempo de compilación, empleando explícitamente, las funciónes **malloc() y free()** respectivamente (Ver Fig. 9).

Hay que recordar que en C no existen mecanismos explícitos como en los lenguajes de POO para la creación automática de clases y el respectivo borrado de las mismas y por lo tanto no se pueden implementar métodos dentro de una Clase de este tipo, por que generaría conflicto, dado que el lenguaje siempre tiene entre sus objetos vinculación (ligadura) de tipo estático.

```
ClaseRio     *Magdalena;
.................
Magdalena = (ClaseRio *) malloc(sizeof(ClaseRio))
.................
free (void* Magdalena);
```

....

Figura 9 Ejemplo de Creación de Objetos

3.6 Plantillas

La reducción de duplicación de código, mecanismo que ya viene inherente en lenguajes del área POO, a través del uso por ejemplo de plantillas (*template*) en C++ o de genéricos (*generic*) en ADA, en C se puede implementar usando *macros*, con el fin de no utilizar tipos, que es el único obstáculo en los lenguajes no propios del área de POO, para la no duplicación de código (Ver Fig. 11).

```
#define Suma (A,B)  (A +B)
..................
..................

int A, B, C;
double D,E, F;
..................
C= Suma(A,B);
....................
F = Suma(D,E);
..................
```

Figura 11 Ejemplo de Plantillas

Hay que recordar que la única parte en donde C no produce ligadura estática con sus objetos (en este caso con sus variables de referencia), es en las macros, lo cual se emplea para generar o emular plantillas.

El empleo y codificación de estas macros, se debe realizar de una forma muy cuidadosa y disciplinada, dado que por la misma no comprobación de tipos podría generar por descuido, desagradables resultados.

3.7 Modularización

El lenguaje C no ofrece un mecanismo explícito para lograr el concepto de Modularización, como por ejemplo el que ofrece JAVA y ADA a través del concepto de Paquete (**Package**), o ***Module*** en Modula, para la descomposción física y lógica de un programa.

La moduralización, en este caso, se hace a través de la estructuración del programa por medio de archivos físicos, los cuales van conteniendo las diferentes clases que se van creando y sus funciones relacionadas (ver Fig. No. 12).

La referencia de una clase a otra cuando se modulariza de esta forma se realiza por medio de la inclusión (#include ....) entre ficheros y los ficheros tipo cabecera (#..........h) (Ver Figura 12)

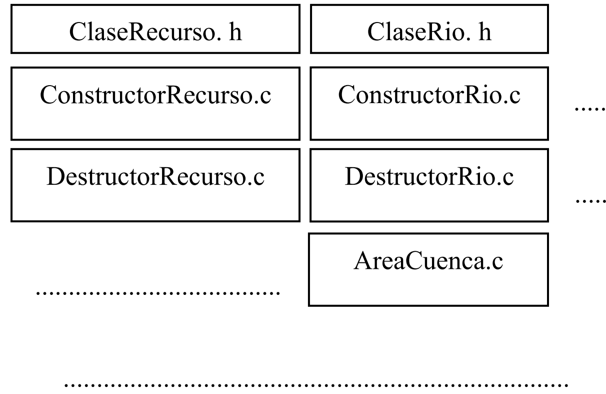

Figura 12 Ejemplo de Modularizaciòn

4. CONCLUSIONES

Como bien puede suponerse, la resolución de métodos en C es estática (ligadura estática), ya que solo ocurre en tiempo de compilación, lo cual no permite el mecanismo de declaración de genéricos (ligadura dinámica), lo que impediría la reducción de código por no duplicación del mismo, salvo lo que se puede realizar, y de una manera mas bien limitada, por medio de macros.

Los equipos de programadores pueden trabajar de manera simultánea en el todo el programa; cada programador trabaja en un archivo diferente que contendría una clase diferente.

Puede ser usado un estilo de programación orientado a objetos. Cada archivo define un tipo particular de objeto como un tipo de dato y las operaciones en ese objeto como funciones. La implementación del objeto puede mantenerse privado al resto del programa. Con lo anterior se logran programas bien estructurados los cuales son fáciles de mantener.

Los archivos pueden contener todas las funciones de un grupo relacionado, por ejemplo todas las operaciones con matrices. Estas pueden ser accedidas como una función de una biblioteca.

Objetos bien implementados o definiciones de funciones pueden ser reusadas en otros programas, con lo que se reduce el tiempo de desarrollo.

En programas muy grandes cada función principal puede ocupar su propio archivo. Cualquier otra función de bajo nivel usada en la implementación puede ser guardada en el mismo archivo, por lo que los programadores que llamen a la función principal no se distraerán por el trabajo de bajo nivel.

Cuando los cambios son hechos a un archivo, solamente ese archivo necesita ser compilado de nuevo

para reconstruir el programa. La herramienta ***make*** de UNIX es muy útil para reconstruir programas con varios archivos, y al mismo tiempo reducir la compilación innecesaria cuando se hace alguna modificación.

Por último, mediante un adecuado diseño y uso de los elementos del lenguaje C, es posible lograr una muy aproximada POO.

4. BIBLIOGRAFÍA

4.1 Referenciada

[1] Rumbaugh James, Blaha Michael, Premerlani, William, Hedí Frederick y Lorensen William, "Modelado y Diseño Orientado a Objetos", Ed. Prentice Hall, España, 1.985, ISBN 013 – 24000698-5.

[2] Rios Patiño, Jorge Ivan, "Administración e inferencia de Bases de Conocimientos usando Bases de Datos relacionales y SQL", Tesis de la Maestría de Ingeniería del Conocimiento, Facultad de Informática, Universidad Politécnica de Madrid, Madrid, España, Abril, 2002.

4.2 Consultada

Brian W Hernigham y Ritchie M. Dennis, "El Lenguaje de Programación en C", Ed. Prentice may Hispanoamericana S.A, México, 1985, ISBN 968-880-024-4.

Booch, Grady, Rambaugh, James y Jacobson, Ivar, , "El Lenguaje Unificado de Modelado: UML, "Editorial Addison Wesley:, Madrid , España, 2000.

Booch, Grady, "Software Design with ADA "Editorial Addison Wesley:, U.S.A , 1.998.