UNIVERSIDAD CENTRAL DE VENEZUELA
FACULTAD DE INGENIERÍA
CICLO BÁSICO
Departamento de Investigación de
Operaciones Y Computación

GUIA DE PROGRAMACIÓN ORIENTADA A OBJETOS EN LENGUAJE C

Elaborada por:
Prof. Belzyt González G.

Caracas, Abril 1997

INDICE DE CONTENIDOS

INTRODUCCIÓN

Son muchas las exigencias que en materia de computación pesan sobre el Ingeniero recién egresado de la Facultad para competir exitosamente en el mercado de trabajo. Estas necesidades no sólo abarcan el dominio de las herramientas básicas de la programación de microcomputadores sino que se extiende a una programación más exigente que domine las estructuras de datos avanzadas, como lo es la programación orientada a objetos. Esta necesidad se hace cada vez mas imperante debido a que nuestros propios estudiantes, al entrar a las escuelas profesionales, se ven en la obligación de utilizar este tipo de técnica. Por esta razón, se propone esta guía, la cual viene facilitar el aprendizaje de una estructura de datos avanzada como los es la programación orientada a objetos, enfocado desde el punto de vista de Clases en Borland C.

OBJETIVO

Esta guía tiene como objetivo que el estudiante adquiera, de forma rápida y sencilla, los conocimientos y destrezas necesarias para resolver problemas complejos utilizando como herramienta el manejo de la programación orientada a objetos.

PROGRAMACIÓN ORIENTADA A OBJETOS
BORLAND C ++

Objeto es una entidad lógica que contiene datos y un código de programación que los manipula. Parte de estos datos o códigos pueden ser privados del objeto e inaccesibles fuera de él. Otra parte puede ser pública y accesible desde cualquier parte del programa. Un objeto en sí, es una variable que debe ser definida como tal por el usuario.

Para crear un objeto en C ++ se debe definir su forma general usando la palabra reservada **class** como se muestra a continuación:

```
class nombre_de_la_clase {
                        datos y funciones privadas
                        public:
                        datos y funciones públicas
                        } lista_de_nombres_objetos;
```

Por defecto todos los elementos definidos en una clase son privados y pueden ser invocados por otros miembros de la misma clase. Todas las partes que están después de la palabra reservada **public** pueden ser llamadas desde otras funciones del programa.

En el ejemplo que se muestra a continuación se crea una clase complejo donde las variables R e I son privadas a la clase mientras que las funciones suma, resta y escribir son funciones miembro público de dicha clase.

```
class complejo {
                int R;
                int I;
                public:
                complejo suma( complejo a, complejo b);
                complejo resta(complejo a, complejo b);
                void escribe(void);
                };
```

Para crear una variable de tipo complejo, se declara al igual que las declaraciones de variables de tipo predefinido, por ejemplo:

```
complejo   s, x;
```

Al momento de escribir la codificación de las funciones miembros de la clase se debe definir a que clase pertenece colocando su nombre después de la definición de tipo de la función seguido por el operador de ámbito (**::**).

```
void complejo :: escribir(void)
{
```

```
        printf("%d + d i", R, I);
}
```

Para ejecutar un a función miembro de una clase, se coloca después del nombre del objeto, el operador punto, seguido del nombre de la función, por ejemplo:

```
x.escribir();
```

Una función miembro de una clase puede llamar a otra función miembro de la misma clase sin utilizar el operador punto. Dos clases diferentes pueden usar funciones miembro con el mismo nombre, ya que al codificarlas se indica a que clase pertenece, por ejemplo:

```
int complejo suma(int a);// función suma de la clase complejo
int lanza suma(int a);// función suma de la clase lanza
```

Herencia: es el proceso por el cual un objeto puede adquirir las propiedades de otro objeto. La herencia permite a una clase incorporar otra clase dentro de su declaración:

```
class clase_derivada: acceso clase_base {
        // cuerpo de la clase derivada
        }
```

Los miembros de la clase derivada tienen acceso a los miembro de la clase base excepto a las partes privadas. El acceso a la clase base puede ser **public** o **private**. Si el acceso es público todas las partes públicas de la clase base son también públicas aquí, si el acceso es privado, las partes públicas y privadas de la clase base, son privadas en la clase derivada.

Si se quiere tener acceso a la parte privada de clase base, desde la clase derivada, se puede declarar **protected**. Mediante esta declaración se logra que la clase derivada tenga acceso a la parte del la clase base que no es pública.

```
class complejo: public simple{
        // cuerpo de la clase complejo
        }
```

Constructores y destructores:

La función constructora es una función especial miembro de la clase, y tiene el mismo nombre de la clase. La función constructora sirve para inicializar las variables miembros de la clase.

```
class complejo {
        int R;
```

```
        int I;
        public:
        complejo (void); //constructor
        void escribe(void);
        };

complejo :: complejo (void) {
        R = I = 0;
        }
```

Nótese que para declarar el constructor no se le antepone el tipo de función, ya que esta es una función especial. Al constructor de un objeto se le invoca al declarar el objeto.

Cuando una clase hereda a otra, hereda también al constructor de la clase base, por lo tanto al declarar un objeto de la clase derivada, se invoca primero al constructor de la clase base y luego al de la clase derivada.

Se puede crear constructores parametrizados, con la idea de inicializar las variables miembros de la clase en valores enviados como parámetros.

```
class complejo {
        int R;
        int I;
        public:
        complejo (void); //constructor
        void escribe(void);
        };

complejo :: complejo (int k, int j) {
        R = I = 0;
        }
```

Como al momento de declarar el objeto se invoca al constructor, es en ese mismo momento que se le envía el (los) parámetro (s) a inicializar.

```
complejo a(2,6);
```

El destructor de una clase se utiliza cuando se necesita realizar alguna función al terminar de utilizar el objeto (al salir de la función donde se declaró), como por ejemplo, liberar memoria dinámica, cerrar archivos, etc.. El destructor tiene el mismo nombre que el constructor pero lo antecede el símbolo ~, por ejemplo:

```
class complejo {
        int R;
        int I;
```

```
        public:
        complejo (void); //constructor
        void escribe(void);
        ~complejo (void); //destructor
        };

complejo :: ~complejo (void) {
        closeall();;
        }
```

Funciones amigas: Son funciones externas a la clase, pero que tienen acceso a la parte privada de la clase. Por lo general se utiliza cuando dos clases comparten la misma función. Para declarar una función amiga, debe hacerse en la declaración de la clase a la que es amiga, como se muestra:

```
class complejo {
        int R;
        int I;
        public:
        friend void amiga(void);
        };
```

Punteros a objetos: En muchas ocasiones se necesita apuntar a una variable de tipo objeto, para esto debemos declarar el puntero de manera que este apunte a un objeto, como se muestra:

```
complejo a, *pob;
```

de esta forma hemos declarado un objeto de nombra a y un puntero a un objeto de la clase complejo, llamado pob. Para utilizar las funciones miembros de la clase del objeto al que se apunta, se utiliza el operados flecha: **->**

```
class complejo {
        int R;
        int I;
        public:
        complejo (void); //constructor
        void escribe(void);
        ~complejo (void); //destructor
        };

main(){
  complejo a, *pob;
```

```
    pob = &ob;
    p -> escribe(); //accesa al miembro escribe a través del puntero
    a.escribe(); //accesa al miembro escribe a través del objeto
}
```

Polimorfismo: permite utilizar un mismo nombre de función para varios propósitos relacionados pero con ligeras diferencias. En C++ se logra el polimorfismo a través de la sobrecarga de funciones.

Sobrecarga de funciones: Se basa en el hecho que dos ó más funciones pueden llevar el mismo nombre, siempre y cuando su declaración de parámetros sea diferente. Por ejemplo, se podría crear dos funciones para elevar un número x al exponente y de manera que una de ellas trabaje con números enteros y devuelva enteros y la otra trabajara con números reales y devuelva reales.

```
int elevar(int x, int y) {
        int t;
        t = exp(y*ln(x));
        elevar = t;
}

float elevar(float x, int y) {
        float t;
        t = exp(y*ln(x));
        elevar = t;
}
```

De esta manera, si el argumento es real llama a la función elevar de datos reales, si el argumento es entero, se llama automáticamente a la función de datos enteros sin tener que memorizar dos nombres de funciones diferentes.

Al igual que las funciones, los constructores también pueden sobrecargarse, cambiando el tipo y la cantidad de parámetros.

Sobrecarga de operadores: al igual que las funciones, C++ permite la sobrecarga de operadores de manera de darle más de un sentido a un operador dependiendo de la naturaleza de los operandos. Casi todos los operadores en C++ pueden ser sobrecargados (+, -, *, /, ++, --, +=, *=, -=, /=, =, etc.), de esta manera se pueden cargar operadores unarios (un operando) y operadores binarios (dos operandos). El único operador que no es posible sobrecargas es el operador ?, el cual trabaja con tres operandos.

Antes de hablar de la sobrecarga de los operadores, es necesario explicar la función del puntero **this**. Cada vez que se invoca a una función miembro, se pasa automáticamente un puntero al objeto que ocasionó el llamado a la función (al objeto que se encuentra a la izquierda del operador punto), para accesar a ese puntero se utiliza la palabra clave **this**.

Para sobrecargar un operador se coloca el tipo de dato que devuelve seguido del nombre de la clase a la que pertenece dicho operador, luego se

coloca el operador de ámbito (::), la palabra clave **operator** y el símbolo del operador que se desea sobrecargar, como se muestra a continuación:

```
tipo clase :: operator # (argumentos)
{ //operación que se desea que realice
...
}
```

Para hacerlo un poco más descriptivo el uso de la sobrecarga de operadores, vea el siguiente ejemplo, el cual crea una clase llamada complejo para sumar y restar números complejos.

```
class complejo {
        int R;
        int I;
        public:
        complejo (int inr, int ini); //constructor
        complejo operator+(complejo t);
        complejo operator-(complejo t);
        complejo operator=(complejo t);
        };


complejo :: complejo (int inr=0, int ini=0)
{
        R = inr;
        I = ini

}


complejo complejo :: operator+(complejo t)
{
complejo temp;
        temp.R = R + t.R;
        temp.I = I + t.I;
        return temp;
}


complejo complejo :: operator-(complejo t)
{
complejo temp;
        temp.R = R - t.R;
        temp.I = I - t.I;
```

```
        return temp;
}
complejo complejo :: operator=(complejo t)
{
complejo temp;
        R = t.R;
        I = t.I;
        return *this;
}

void complejo :: escribe (void)
{
        printf (" %d + %d i);
}

main()
{
    complejo a(4,8), b(5,3),c,d;

        c = a + b;
        d = a - b;
        a.escribe();
        b.escribe();
        c.escribe();
        d.escribe();
}
```

En este ejemplo se observa que al sobrecargar los operadores hemos logrado sumar y restar números complejos con los operadores de suma y resta que manejamos naturalmente. Para un operador binario como la suma y la resta se trabaja de manera que el resultado de la operación no modifique ninguno de sus operandos, es decir cuando sumamos a + b, no pretendemos que el resultado se guarde en a ni en b, por esta razón utilizamos una variable auxiliar temp para almacenar el valor de la suma. El objeto que se encuentra a la izquierda del operador es el que lo invoca, por lo cual estará apuntado por this. El objeto que se encuentra a la derecha del operador se trata como parámetro de este.

Para operadores binarios como la igualdad, se desea que se modifique el valor del objeto que se encuentra a la izquierda de este, por esta razón se devuelve el puntero this ya que es este objeto el que ocasiona el llamado del operador. El objeto a la derecha ingresa como parámetro al operador.

BIBLIOGRAFÍA RECOMENDADA

- Atkinson, L. Atkinson, M. **Using Borland C++3**. Que Corporation. Programming Series, Carmel 1992
- Perry, G. **Aprendiendo Programación Orientada a Objetos con Turbo C++**. Prentice-Hall Hispanoamericana, Mexico 1993
- Norton, P. Yao, P. **Borlnd C++ Progamming for Windows**. Bantam Books, New York 1992
- Deitel, H.M. Deitel, P.J. **Como Programar en C/C++.** Prentice-Hall Hispanoamericana, Mexico 1995