

# Manual 5 - 2do Torneo de Programación Competitiva

Lions R.C.

Agosto 2019

## Contents

<b>1</b>	<b>Programación orientado a objetos</b>	<b>2</b>
1.1	Uniones . . . . .	2
1.2	Clases . . . . .	3
1.3	Constructores y destructores . . . . .	6
1.4	Sobrecargamiento . . . . .	9
1.5	Sobrecargamiento de operadores . . . . .	11
1.6	Clases heredadas . . . . .	12
1.7	Variables estáticas . . . . .	13
<b>2</b>	<b>Punteros</b>	<b>15</b>
2.1	Referencias . . . . .	15
2.2	Punteros a objetos . . . . .	15
2.3	Puntero this . . . . .	15
<b>3</b>	<b>Asignación de memoria</b>	<b>15</b>
<b>4</b>	<b>Estructuras y grafos con punteros</b>	<b>15</b>
4.1	Listas encadenadas, dobles y cíclicas . . . . .	15
4.2	Arboles de binario . . . . .	15
4.3	Algoritmo de Kruskal . . . . .	15
4.4	Algoritmo de Prim . . . . .	15
4.5	Arboles de expansión . . . . .	15
<b>5</b>	<b>Trucos de programación competitiva</b>	<b>15</b>
5.1	Arboles de segmentos . . . . .	15
5.2	Tablas hash . . . . .	15
5.3	Números grandes . . . . .	15
5.4	Conversiones de bases . . . . .	15
<b>6</b>	<b>Enums</b>	<b>15</b>
<b>7</b>	<b>Manipulación de bits</b>	<b>15</b>

<b>8 Matemáticas modulares</b>	<b>15</b>
8.1 Adición . . . . .	15
8.2 Multiplicación . . . . .	15
8.3 Exponenciación . . . . .	15



## 1 Programación orientado a objetos

Como hemos visto anteriormente es conveniente trabajar con grupos de datos en structs en lugar de guardar los datos en pares de pares o tenerlos sueltos. Cada instancia de un struct se puede conocer como un objeto, todos los objetos tienen la característica de ser conformado por distintos datos y se pueden crear diferentes instancias de un objeto con valores distintos. Un objeto puede ser visto como una base que puede ser duplicado y que mantiene sus datos de manera estructurada.

Como hemos visto con los structs cada objeto puede ser instanciado o destruido (esto tiende a ocurrir después de que termina nuestro programa), y cada objeto puede ser declarado con un nombre único.

### 1.1 Uniones

La unión es un objeto parecido a un struct con la única diferencia que solo permite que el usuario guarde un dato en todas sus variables en un cierto tiempo para ahorrar espacio de memoria. Esta estructura siempre ocupará el tamaño de la variable más grande de la lista de todas sus variables.

Por ejemplo, si se tiene un int, un float y un long long int en un struct este ocuparía un espacio de 16 bytes para guardar los 4 bytes del int, los 4 bytes del float y los 8 bytes del long long int, pero una union con estas tres variables solo ocupará 8 bytes porque este es el tamaño del dato más grande (el long long int).

Una desventaja de este tipo de dato es que no se puede saber cuál es la variable que está siendo guardado, así que se debe usar otra variable o tener un contexto específico. Debido a estas limitaciones la union es una estructura raramente usada.

Aquí se presenta el ejemplo de un código que almacena una dirección (arriba, abajo, izquierda o derecha). Como siempre se tendrán dos arreglos distintos

con uno para guardar direcciones horizontales y uno que guarda direcciones verticales, entonces se puede crear una union que guarda arriba o abajo en un booleano o derecha o izquierda en otra:

Listing 1: Uniones

```
#include <iostream>

using namespace std;

union Direccion {
    bool arriba;
    bool derecha;
};

int main() {
    Direccion verticales[4];
    Direccion horizontales[4];
    for(int i = 0; i < 4; i++) {
        cin >> verticales[i].arriba;
    }
    for(int i = 0; i < 4; i++) {
        cin >> horizontales[i].derecha;
    }
    for(int i = 0; i < 4; i++) {
        if(verticales[i].arriba) {
            cout << "arriba" << endl;
        } else {
            cout << "abajo" << endl;
        }
    }
    for(int i = 0; i < 4; i++) {
        if(horizontales[i].derecha) {
            cout << "derecha" << endl;
        } else {
            cout << "izquierda" << endl;
        }
    }
}
```

## 1.2 Clases

Las clases son objetos especiales que te permiten guardar datos como un struct, pero además pueden tener funciones internas y variables privadas. Una variable pública es una que puede ser modificada desde afuera de la clase mientras que una privada tiene que ser modificada desde una llamada interna. Para definir

la privacidad de una clase se debe escribir **public:** antes de la lista de las variables y funciones públicas y **private:** antes de la lista de funciones y variables privadas.

Por ahora solo crearemos una clase con variables públicas utilizando una estructura parecida a un struct:

Listing 2: Declarando clases

```
#include <iostream>

using namespace std;

class Cliente {
    public:
        string nombre;
        float dinero;
};

int main() {
    Cliente pablo;
    pablo.nombre = "Pablo_Cesar";
    pablo.dinero = 12.1;
    cout << pablo.nombre << "_tiene_" << pablo.dinero;
}
```

Algo que habíamos mencionado es que las clases pueden tener sus propios funciones, así que crearemos uno para ver cuanto dinero tiene cada cliente y otro para sumarle una cantidad de dinero a su cuenta:

Listing 3: Funciones internas

```
#include <iostream>

using namespace std;

class Cliente {
    public:
        string nombre;
        float dinero;

        void imprimeSaldo() {
            cout << nombre << "_tiene_" << dinero << endl;
        }

        void deposita(float cantidad) {
            dinero += cantidad;
        }
};
```

```

int main() {
    Cliente pablo;
    pablo.nombre = "Pablo_Cesar";
    pablo.dinero = 12.1;
    Cliente jose;
    jose.nombre = "Jose_Miguel";
    jose.dinero = 135.23;
    jose.imprimeSaldo();
    pablo.imprimeSaldo();
    pablo.deposita(300);
    pablo.imprimeSaldo();
}

```

Podemos ver que ambos clientes tienen las mismas funciones pero cada función es específica a cada cliente, una llamada a `imprimeSaldo` solo imprime el saldo de ese cliente y no de todos.

La razón por la que existen las variables privadas es para asegurar que esta variable solo esté modificada correctamente. Si fuéramos a llamar a `deposita` con un número negativo se perdería dinero así que es recomendable hacer unas pruebas dentro de esta función, hacer la variable `dinero` privada y crear otras funciones que manejen esta variable.

Este concepto se llama la abstracción de datos debido a que estamos protegiéndolos de uso equivocado o indeseado. Si intentamos leer o modificar la variable `dinero` desde `main` se nos arrojará un error, mientras que en cualquiera de las funciones de nuestra clase no habrá problema. Nuestro código quedará de la siguiente manera:

Listing 4: Abstracción de datos

```

#include <iostream>

using namespace std;

class Cliente {
    private:
        float dinero;

    public:
        string nombre;

        void imprimeSaldo() {
            cout << nombre << "_tiene_ $" << dinero << endl;
        }

        void deposita(float cantidad) {
            if(cantidad <= 0) {

```

```

        cout << "Deposito_invalido" << endl;
    } else {
        dinero += cantidad;
    }
}

};

int main() {
    Cliente pablo;
    pablo.nombre = "Pablo_Cesar";
    pablo.deposita(12.1);
    Cliente jose;
    jose.nombre = "Jose_Miguel";
    jose.deposita(135.23);
    jose.imprimeSaldo();
    pablo.imprimeSaldo();
    pablo.deposita(-300);
    pablo.imprimeSaldo();
}

```

### 1.3 Constructores y destructores

Si vemos el ejemplo anterior ahora hemos protegido nuestro dinero de movimientos indeseados pero no podemos darle un valor inicial a nuestro dinero sin llamar a `deposita`, y puede ser que no queremos usar esa función para inicializar nuestro dinero.

Existe una función especial llamada el constructor que puede recibir varios parametros y que puede ser llamada cuando se instancializa un nuevo objeto.

El constructor siempre tiene el mismo nombre que el nombre de la clase y es una función sin tipo. En esta función es recomendable inicializar todas las variables de una clase:

Listing 5: Constructores

```

#include <iostream>

using namespace std;

class Cliente {
private:
    float dinero;

public:
    string nombre;

    //constructor

```

```

    Cliente(string nombreInicial, float saldoInicial) {
        nombre = nombreInicial;
        if(saldoInicial < 0) {
            dinero = 0;
        } else {
            dinero = saldoInicial;
        }
    }

    void imprimeSaldo() {
        cout << nombre << "_tiene_" << dinero << endl;
    }

    void deposita(float cantidad) {
        if(cantidad <= 0) {
            cout << "Deposito_invalido" << endl;
        } else {
            dinero += cantidad;
        }
    }
};

int main() {
    Cliente pablo("Pablo_Cesar", -22.37);
    Cliente jose("Jose_Miguel", 135.23);
    jose.imprimeSaldo();
    pablo.imprimeSaldo();
    pablo.deposita(12.1);
    pablo.imprimeSaldo();
}

```

Podemos ver que también nos ayuda a simplificar el código que escribimos dentro de la función main.

También existen los destructores que son funciones especiales que se corren a la hora de borrarse una clase. Una clase solo se tiende a borrar al final de su ejecución, pero más adelante veremos otros casos donde se pueden borrar manualmente.

Para declarar el destructor se debe escribir una función sin parametros y sin tipos con un tilde seguido por el nombre de la clase.

Listing 6: Destructores

```

#include <iostream>

using namespace std;

class Cliente {

```

```

    private:
        float dinero;

    public:
        string nombre;

        //constructor
        Cliente(string nombreInicial, float saldoInicial) {
            nombre = nombreInicial;
            if(saldoInicial < 0) {
                dinero = 0;
            } else {
                dinero = saldoInicial;
            }
            cout << "Cliente_" << nombre << "_creado" << endl;
            imprimeSaldo();
        }

        //destructor
        ~Cliente() {
            cout << "Cliente_" << nombre << "_eliminado" << endl;
        }

        void imprimeSaldo() {
            cout << nombre << "_tiene_" << dinero << endl;
        }

        void deposita(float cantidad) {
            if(cantidad <= 0) {
                cout << "Deposito_invalido" << endl;
            } else {
                dinero += cantidad;
                cout << nombre << "_ha_recibido_" << cantidad << endl;
            }
        }
    };

    int main() {
        for(int i = 0; i < 3; i++) {
            string nombre;
            float saldo;
            cout << "Nombre_del_cliente:_" << endl;
            cin >> nombre;
            cout << "Saldo_inicial:_" << endl;
            cin >> saldo;
            Cliente temporal(nombre, saldo);

```



```

        float deposito;
        cout << "Dinero a depositar:_" << endl;
        cin >> deposito;
        temporal.deposita(deposito);
        temporal.imprimeSaldo();
        cout << endl;
    }
}

```

[Liga al código](#)

## 1.4 Sobrecargamiento

En tanto las funciones regulares como las funciones de clases se puede llevar a cabo el sobrecargamiento. Esto consiste en tener varias funciones con el mismo nombre pero con diferentes tipos de parámetros. Por ejemplo, si quisiéramos tener una función suma que es capaz de sumar dos números o dos dígitos en forma de caracteres ('5' + '6' nos daría 11), entonces podemos hacer dos funciones que se sobrecargan:

Listing 7: Sobrecargamiento

```

#include <iostream>

using namespace std;

int suma(int a, int b) {
    return a + b;
}

int suma(char a, char b) {
    return (a - '0') + (b - '0');
}

int main() {
    cout << suma(5, 6) << endl;
    cout << suma('5', '6') << endl;
    cout << suma('9', '7') << endl;
}

```

Aquí podemos ver que la función suma es sobrecargada y que la primera vez que se llama utiliza la función de arriba porque sabe que los parámetros son dos enteros, mientras que las otras dos veces corre la función de abajo porque los parámetros son caracteres.

El sobrecargamiento también permite la llamada de la misma función, así que se puede crear una función general que es llamada por las funciones que sobrecargan:

Listing 8: Sobrecargamiento

```
#include <iostream>

using namespace std;

int potencia(int a, int b) {
    int resultado = 1;
    for(int i = 0; i < b; i++) {
        resultado *= a;
    }
    return resultado;
}

int potencia(char a, char b) {
    return potencia(a - '0', b - '0');
}

int main() {
    cout << potencia(5, 6) << endl;
    cout << potencia('5', '6') << endl;
    cout << potencia('9', '7') << endl;
}
```

Para una clase el sobrecargamiento es exactamente igual:

Listing 9: Sobrecargamiento de clases

```
#include <iostream>

using namespace std;

class Cliente {
public:
    string nombre;
    float dinero;

    Cliente(string nombreInicial) {
        nombre = nombreInicial;
        dinero = 0;
    }

    Cliente(string nombreInicial, float dineroInicial) {
        nombre = nombreInicial;
        dinero = dineroInicial;
    }
};
```

```

int main() {
    Cliente pablo("Pablo");
    Cliente jorge("Jorge", 121.35);
}

```

## 1.5 Sobrecargamiento de operadores

Se pueden sobrecargar los operadores principales (+, -, =, ...) para que puedan manipular las instancias de dos clases. Digamos que tenemos una clase que sirve como un punto bidimensional y quisieramos que se sumarán dos puntos con el operador +, entonces se puede sobrecargar este operador en nuestra clase de la siguiente manera:

Listing 10: Sobrecargamiento de operadores

```

#include <iostream>

using namespace std;

class Punto {
    public:
        int x;
        int y;

        Punto(int xInicial, int yInicial) {
            x = xInicial;
            y = yInicial;
        }

        void imprimePunto() {
            cout << "(" << x << ", " << y << ")" << endl;
        }

        Punto operator+(const Punto& p) {
            Punto suma(x + p.x, y + p.y);
            return suma;
        }
};

int main() {
    Punto a(6, 11);
    Punto b(-3, 25);
    Punto c = a + b;
    c.imprimePunto();
}

```

Se debe crear una función con nombre `operator(signo)` y este debe regresar o modificar la clase que se esta utilizando, como parámetro debe recibir un **const tipo& nombre** donde esta variable es la otra variable que se esta sumando, restando o modificando.

## 1.6 Clases heredadas

Suele suceder que ocupas crear varias clases especificas con diferencias pequeñas. Para evitar tener que copiar el mismo código muchas veces se puede crear una clase general que contiene variables y funciones que se comparten entre otras clases especificas que "heredan" esta clase general.

Para heredar una clase se debe escribir dos puntos, `public` y el nombre de la clase general despues de la clase que hereda estas propiedades. Para llamar el constructor de una clase heredada se debe hacer algo parecido donde se pone dos puntos, el nombre del constructor de la clase que hereda y las variables que se desean pasar a ese constructor original.

Aquí podemos ver un ejemplo de un código donde se tiene dos clases, la clase `Miembro` contiene todas las funciones y variables de la clase `Cliente`, pero le hacemos unas modificaciones para que solo miembros reciben ciertas ventajas. Por ejemplo, no se puede modificar la variable `puntos` de un cliente porque esta clase no la tiene, pero la variable `nombre` si es igual para las dos clases.

Listing 11: Heredación

```
#include <iostream>

using namespace std;

class Cliente {
    public:
        string nombre;
        double dinero;

        Cliente(string nombreInicial) {
            nombre = nombreInicial;
            dinero = 0;
        }

        void venderArticulo(double precio) {
            dinero += precio;
            cout << nombre << " _no_recibio_puntos
            por_vender_un_articulo_porque_no_es
            miembro" << endl;
        }

        void verDatos() {
            cout << nombre << " _tiene_" << dinero << endl;
        }
    };
};
```

```

    }
};

class Miembro : public Cliente {
    private:
        double puntos;

    public:
        Miembro(string nombreInicial) : Cliente(nombreInicial) {
            puntos = 0;
        }

        void venderArticulo(double precio) {
            double puntosDeVenta = precio / 10;
            puntos += puntosDeVenta;
            dinero += precio;
            cout << nombre << "_obtuvo_" << puntosDeVenta <<
                "_puntos_por_vender_un_articulo_porque_es_miembro" << endl;
        }

        void verDatos() {
            cout << nombre << "_tiene_" << dinero <<
                "_y_" << puntos << "_puntos" << endl;
        }
};

int main() {
    Cliente pablo("Pablo");
    Miembro jose("Jose");
    pablo.venderArticulo(653.12);
    jose.venderArticulo(653.12);
    pablo.verDatos();
    jose.verDatos();
}

```

Si observamos este código podemos ver que Miembro es lo mismo que Cliente pero con otra variable y dos funciones sobrecargadas.

## 1.7 Variables estáticas

Las variables estáticas son especiales debido a que solo existe una instancia de ellas a la vez. Si declaramos una clase con una variable estática, esta variable siempre se inicializará en cero y sera igual para todas las clases.

Para crear una variable estática se debe escribir **static** antes del nombre de la variable y se debe declarar esta variable fuera de la clase usando **Clase::nombre**;

Listing 12: Variables estáticas

```
#include <iostream>

using namespace std;

class Punto {
public:
    static int puntosMarcados;
    int x;
    int y;

    Punto(int xInicial, int yInicial) {
        x = xInicial;
        y = yInicial;
        puntosMarcados++;
    }
};

int Punto::puntosMarcados;

int main() {
    Punto a(3, 5);
    Punto b(5, 9);
    Punto c(2, 2);
    Punto d(1, 4);
    cout << "Hay_" << Punto::puntosMarcados << "_puntos" << endl;
}
```

Si corremos este ejemplo se imprimirán que hay 4 puntos porque el constructor de cada punto le sumó 1 a la variable puntosMarcados.

- 2 Punteros**
  - 2.1 Referencias
  - 2.2 Punteros a objetos
  - 2.3 Puntero this
- 3 Asignación de memoria**
- 4 Estructuras y grafos con punteros**
  - 4.1 Listas encadenadas, dobles y cíclicas
  - 4.2 Arboles de binario
  - 4.3 Algoritmo de Kruskal
  - 4.4 Algoritmo de Prim
  - 4.5 Arboles de expansión
- 5 Trucos de programación competitiva**
  - 5.1 Arboles de segmentos
  - 5.2 Tablas hash
  - 5.3 Números grandes
  - 5.4 Conversiones de bases
- 6 Enums**
- 7 Manipulación de bits**
- 8 Matemáticas modulares**
  - 8.1 Adición
  - 8.2 Multiplicación
  - 8.3 Exponenciación