Manual 5 - 2do Torneo de Programación Competitiva

Lions R.C.

Agosto 2019

Contents

1 Programación oriendado 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 caracteristica 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 instancializado o destruido (esto tiende a ocurrir despues de que termina nuestro programa), y cada objeto puede ser declarado con un nombre unico.

1.1 Uniones

La unión es un objeto parecido a un struct con la unica 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 mas grande (el long long int).

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

Aqui se presenta el ejemplo de un código que almacena una dirección (arriba, abajo, izquierda o derecha). Como siempre se tendran 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 internos 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 habiamos mencionado es que las clases pueden tener sus propios funciones, asi 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 funcion es especifico 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 este modificada correctamente. Si fueramos a llamar a deposita con un número negativo se perdería dinero asi que es recomendable hacer unas pruebas dentro de esta función, hacer la variable dinero privada y crear otras funciones que manejan esta variable.

Este concepto se llama la abstracción de datos debido a que estamos protegiendolos 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.

Tambien 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